PolyPublie
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Column Generation in Machine Learning
Auteur: Author:	Krunal Kishor Patel
Date:	2024
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Patel, K. K. (2024). Column Generation in Machine Learning [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/58317/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/58317/
Directeurs de recherche: Advisors:	Guy Desaulniers, & Andrea Lodi
Programme: Program:	Doctorat en mathématiques

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Column Generation in Machine Learning

KRUNAL KISHOR PATEL
Département de mathématiques et de génie industriel

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Mathématiques

Avril 2024

© Krunal Kishor Patel, 2024.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

Column Generation in Machine Learning

présentée par **Krunal Kishor PATEL**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Louis-Martin ROUSSEAU, président
Guy DESAULNIERS, membre et directeur de recherche
Andrea LODI, membre et codirecteur de recherche
Quentin CAPPART, membre
Stefano GUALANDI, membre externe

DEDICATION

To those who cheered me on with every step I took,
And to the doubters, your skepticism was the perfect hook.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisors, Prof. Andrea Lodi and Prof. Guy Desaulniers, for their invaluable guidance, support, and encouragement throughout my PhD journey. Their expertise and mentorship have been instrumental in shaping this research. Special thanks to Andrea for making extra efforts to get me started with the Ph.D. program at Polytechnique Montreal.

I am grateful to Mehdi Taobane and Dr. Khalid Laaziri from the Canada Excellence Research Chair (CERC) for their assistance and support during my time at CERC. Mehdi's facilitation of communication and Khalid's provision of computational resources were essential for the success of this research.

Special thanks to my parents for their unwavering love and encouragement, as well as to my friends from my undergrad studies and school for their emotional support throughout this endeavor. Also thanks to my fellow PhD students, postdocs, and employees working nearby me for their camaraderie and support during my time at CERC.

I would also like to acknowledge the members of the chess club "PolyÉchecs" for the enjoyable chess games and moments of relaxation during my time at Polytechnique.

I would also like to thank the committee for spending their valuable time to read my thesis and provide useful feedback.

Finally, I am grateful to all those who have contributed to my academic and personal growth, even if their names are not mentioned here.

Funding for this research was provided by the Canada Excellence Research Chair (CERC).

RÉSUMÉ

Cette thèse explore l'intrégation de la recherche opérationnelle et de l'apprentissage automatique, mettant l'accent sur l'augmentation de la précision et de la praticalité au sein des cadres d'apprentissage automatique (ML) grâce à l'intégration stratégique d'algorithmes de génération de colonnes. Malgré l'ubiquité des algorithmes ML dans les applications de reconnaissance de motifs, leur manque d'interprétabilité et leur dépendance aux heuristiques introduisent de l'imprécision, soulignant ainsi la nécessité de méthodologies plus affinées.

Les objectifs de la recherche sont doubles : d'abord, examiner l'application de classificateurs basés sur la génération de colonnes dans le domaine du ML, et ensuite, résoudre les défis pragmatiques entravant l'intégration fluide de ces classificateurs dans des scénarios réels. L'exploration englobe le développement d'algorithmes de génération de colonnes pour des applications pratiques, l'amélioration des algorithmes heuristiques pour l'apprentissage d'arbres de classification, et le développement d'un cadre logiciel facilitant la mise en œuvre efficace de modèles ML basés sur la génération de colonnes.

Nous présentons une revue exhaustive de la littérature (Chapitre 2) qui recense divers classificateurs basés sur la génération de colonnes en apprentissage automatique et propose des observations communes à ce domaine. Notre première contribution (Chapitre 3) étend l'application de la méthode de génération de colonnes pour générer des règles de décision booléennes à des situations du monde réel pour classer les NOTAM, abordant spécifiquement les défis posés par la classification multiclasse à travers la classification un-contre-tous. Nous introduisons une heuristique exploitant un solveur CP-SAT pour accélérer la génération de colonnes, ainsi qu'une méthode pour réduire l'espace de recherche dans le sous-problème.

Dans notre contribution suivante (Chapitre 4), l'accent se déplace vers l'amélioration des heuristiques basées sur la génération de colonnes pour l'apprentissage d'arbres de classification. Des modifications substantielles au modèle de sous-problème sont présentées pour réduire significativement le nombre de sous-problèmes dans des instances de classification multiclasse. Le chapitre démontre l'utilisation de contraintes dépendantes des données dans le problème maître en tant que plans coupants, introduisant un modèle de séparation pour identifier les violations de contraintes dans la solution de la relaxation linéaire. De plus, une routine de prétraitement et d'initialisation est décrite pour réduire la taille du problème maître et des sous-problèmes, réduisant ainsi les temps d'entraînement.

Notre prochaine contribution (Chapitre 5) dévoile un cadre logiciel conçu pour simplifier la conception et la mise en œuvre de modèles ML basés sur la génération de colonnes. Le cadre

vise à fournir une implémentation efficace, flexible, bien documentée et en open source des composants communs utilisés dans divers classificateurs basés sur la génération de colonnes.

Dans notre étude, nous découvrons un défi significatif au sein des classificateurs basés sur la génération de colonnes, comme discuté dans les Chapitres 3 et 4: une part substantielle du processus d'entraînement est consacrée à la résolution de sous-problèmes, ce qui entraîne un goulot d'étranglement dans l'efficacité globale. Ces sous-problèmes, composés de MIPs qui diffèrent principalement dans leur fonction objectif entre les itérations des génération de colonnes, sont résolus à partir de zéro. Cependant, dans le Chapitre 6, nous proposons des techniques innovantes pour exploiter les similarités inhérentes entre ces MIPs. Notre approche implique la réutilisation d'informations essentielles obtenues à partir de la résolution d'instances précédentes, conduisant à des améliorations substantielles dans les instances ultérieures de la séquence. Notre approche de réutilisation d'informations pour résoudre des MIPs similaires en séquence est reconnue pour son excellence dans la compétition computationnelle de l'atelier MIP 2023.

ABSTRACT

This thesis delves into the convergence of operations research and machine learning (ML), emphasizing the augmentation of precision and practicality within ML frameworks through the strategic integration of column generation (CG) methodologies. Despite the ubiquity of ML algorithms in pattern recognition applications, they lack interpretability, and their reliance on heuristics introduces imprecision, underscoring the need for more refined methodologies.

The research objectives are two-fold: first, to investigate the application of CG-based classifiers in the ML domain, and second, to address the pragmatic challenges hindering the seamless integration of these classifiers into real-world scenarios. The exploration involves extending CG methodologies to pragmatic applications, refining heuristic algorithms for learning classification trees, and developing a software framework facilitating the efficient implementation of CG-based ML models.

We present a comprehensive literature review (Chapter 2) that serves as the foundational exploration of various CG-based classifiers in ML, elucidating common observations in the field. Our first work (Chapter 3) extends the application of CG-based methods for generating boolean decision rules to real-world instances for classifying NOTAMs, specifically addressing challenges posed by multiclass classification through one-vs-rest classification. We introduce a heuristic leveraging a CP-SAT solver to expedite CG and a method to reduce the search space in the subproblem.

In our next contribution (Chapter 4), the focus shifts to enhancing CG-based heuristics for learning classification trees. Substantial modifications to the subproblem model are presented to reduce the number of subproblems in multiclass classification instances significantly. The chapter demonstrates the use of data-dependent constraints in the master problem as cutting planes, introducing a separation model to identify violations of constraints in the LP relaxation solution. Furthermore, a preprocessing and initialization routine is outlined to reduce the size of the master problem and subproblems, consequently reducing training times.

Our next contribution (Chapter 5) unveils a software framework created to simplify the design and implementation of CG-based ML models. The framework aims to provide an efficient, flexible, well-documented, and open-sourced implementation of common components used in various CG-based classifiers.

In our study, we uncover a significant challenge within CG-based classifiers, as discussed in Chapters 3 and 4: a substantial portion of the training process is devoted to resolving

subproblems, leading to a bottleneck in the overall efficiency. These subproblems, consisting of MIPs that differ mainly in their objective function between CG iterations, are solved from scratch. However, in Chapter 6, we propose innovative techniques to exploit the inherent similarities among these MIPs. Our approach involves reusing essential information obtained from solving previous instances, leading to substantial improvements in later instances within the sequence. Our approach for reusing information to solve similar MIPs in sequence is recognized for its excellence in the computational competition of the MIP workshop 2023.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

RÉSUMÉ v

ABSTRACT vii

TABLE OF CONTENTS ix

LIST OF TABLES xii

LIST OF FIGURES xiv

LIST OF SYMBOLS AND ACRONYMS xv

LIST OF APPENDICES xvi

CHAPTER 1 INTRODUCTION 1
1.1 Background 2
1.1.1 Column Generation 2
1.1.2 Machine Learning 3
1.2 Research Objectives 4
1.3 Thesis Outline 6

CHAPTER 2 LITERATURE REVIEW 7
2.1 Binary Classification using Boolean Decision Rules 7
2.2 Decision Trees 11
2.3 Linear Support Vector Machines 15
2.4 Boosting 19
2.5 Common Observations 21

CHAPTER 3 ARTICLE 1: EXPLAINABLE PREDICTION OF QCODES FOR NOTAMS USING COLUMN-GENERATION 23
3.1 Introduction 23
3.2 Overview of Boolean Decision Rules and Other Classification Methods 26

3.3 Extending the Model for Multiclass Classification 30
3.3.1 Change in the Objective Function 30
3.3.2 Candidate Weights 31
3.4 Computational Approach 32
3.4.1 Data Preprocessing 32
3.4.2 Training 33
3.4.3 Output 35
3.5 Computational Results 35
3.5.1 Tuning Experiments 35
3.5.2 Comparison with Standard ML Models 40
3.5.3 Practical Impact 42
3.6 Conclusion and Future Work 43

CHAPTER 4 ARTICLE 2: AN IMPROVED COLUMN-GENERATION-BASED MATHEURISTIC FOR LEARNING CLASSIFICATION TREES 45
4.1 Introduction 45
4.2 Overview of [1] Column Generation Approach 47
4.3 Modifications of the Column Generation Approach 52
4.3.1 Merged SPs 52
4.3.2 Redundancy of MP Constraints (4.1c) 54
4.3.3 Search for Violated MP Constraints (4.1c) 55
4.4 Computational Results 57
4.4.1 Datasets and Experimental Setup 57
4.4.2 Initialization and Preprocessing Results 60
4.4.3 Merged SPs Results 61
4.4.4 MP Constraints (4.1c) Results 61
4.4.5 Comparison with [1] 66
4.5 Conclusion and Future Work 67

CHAPTER 5 A FRAMEWORK FOR COLUMN-GENERATION-BASED CLASSIFIERS IN SCIKIT-LEARN 71
5.1 Introduction 71
5.2 Framework Design 72
5.2.1 Master Problem 72
5.2.2 Subproblem 73
5.2.3 Classifier 74
5.3 Workflow 75

5.3.1 Initialization . . . 75
5.3.2 Training . . . 76
5.3.3 Prediction . . . 77
5.4 Examples . . . 77
5.4.1 Boolean Decision Rule Generation . . . 77
5.4.2 Decision Tree . . . 78
5.5 Exploiting Similarities in the Subproblems . . . 79

CHAPTER 6 ARTICLE 3: PROGRESSIVELY STRENGTHENING AND TUNING MIP SOLVERS FOR REOPTIMIZATION . . . 81
6.1 Introduction . . . 81
6.2 Benchmarking . . . 83
6.3 Reusing Primal Information . . . 85
6.4 Reusing and Improving Branching Information . . . 88
6.4.1 Reusing Branching Information . . . 88
6.4.2 Improving Pseudocosts . . . 90
6.5 Automated Parameter Tuning . . . 91
6.6 Other Optimizations . . . 97
6.7 All Together . . . 98
6.8 Conclusion and Future Work . . . 100

CHAPTER 7 CONCLUSION . . . 103
7.1 Summary of Works . . . 103
7.2 Future Research . . . 104

REFERENCES . . . 105

APPENDICES . . . 115

LIST OF TABLES

Table 2.1 Notations for the MIP (2.1). 9
Table 2.2 Notations for the MIP (2.2). 10
Table 2.3 Notation for the MIP (2.3). 13
Table 2.4 Additional notation for the SP (2.4) defined for leaf l and target t. . . 15
Table 2.5 Interpretation of ξ values 16
Table 3.1 Model description: sets, parameters, and variables. 27
Table 3.2 Subproblem description: sets, parameters, and variables. 28
Table 3.3 Hyperparameter tuning experimental results 36
Table 3.4 Hyperparameter tuning training statistics 37
Table 3.5 Accuracy for the hyperparameter configuration ($P = 4, C = 30, I = 30, N = 20$) by using different candidate list lengths K. The results are compared with the last line of Table 3.3 denoted as $K = \infty$. . . . 38
Table 3.6 Results for best hyperparameter values. The last three lines consider only the Qcodes with at least 50 examples. 39
Table 3.7 Weights for clauses for the boolean decision rule for label 'LAAS' . . 40
Table 3.8 Hyperparameter tuning for the standard ML algorithms. 41
Table 3.9 Results of different ML models for multiclass classification. 43
Table 4.1 Notation for the MIP (4.1). 50
Table 4.2 Additional notation for the SP (4.2) defined for leaf l and target t. . . 51
Table 4.3 Additional notation for the SP (4.3) for leaf l. 53
Table 4.4 Notation for the separation model (4.8). 58
Table 4.5 Dataset sizes. 59
Table 4.6 Average training accuracy gain over CART for different ways of using constraints (4.1c) with a 600s time limit. 65
Table 4.7 Comparision of training accuracy gains for small datasets. 69
Table 4.8 Comparision of testing accuracy gains for the large datasets. 70
Table 4.9 Training accuracy gains of the revised CG heuristic for the large datasets. 70
Table 5.1 Tuned parameters for the CP-SAT solver. 79
Table 6.1 Changing components and time limits of different series. 84
Table 6.2 Percentages of instances for which the provided hints are converted into at least one feasible solution by *completesol* heuristic for different types of hints. The highest numbers are in bold. 88

Table 6.3 Effects of providing solution hints on total scores. Average BASE scores and % improvements caused by SOLHINT variant. Positive improvement numbers are in bold. 89
Table 6.4 Effects of reusing branching history on total scores. Average BASE scores and % improvements caused by REHIS and REHIS+SB variants. Better positive improvement numbers are in bold. 91
Table 6.5 Convergence comparison between C/N_v and $C/\sqrt{N_v}$. 93
Table 6.6 Selected parameter values during deterministic exploration. 94
Table 6.7 Selected values and count for the tuned parameters for each series. . . 96
Table 6.8 Effects of automated parameter tuning on total scores. Average NO-TUNING scores and % improvements caused by TUNING variant. Positive improvement numbers are in bold. 97
Table 6.9 Effects of automated parameter tuning on total scores. Average NO-TURNOFF scores and % improvements caused by TURNOFF variant. Positive improvement numbers are in bold. 99
Table 6.10 Average total scores of different techniques used for reoptimization. . 100
Table 6.11 Average total scores of reoptimization techniques per batch. 100
Table 6.12 Shifted geometric means of solving time for instances for each series. . 101
Table Table A.1 Effects of providing solution hints on total scores. Best scores are in bold. 115
Table Table A.2 Effects of reusing branching history and strong branching for the first instance on total scores. Best scores are in bold. 117
Table Table A.3 Effects of automated parameter tuning on total scores. Best scores are in bold. 120
Table Table A.4 Effects of turning off inefficient solver parts on total scores. Best scores are in bold. 123
Table A.5 Aggregated results for changing components. 126
Table A.6 Aggregated results for series grouped by geometric means of solving time in BASE. 126
Table A.7 Aggregated results for series grouped by average BASE scores. . . . 126

LIST OF FIGURES

Figure 3.1	A typical NOTAM for London Heathrow Airport.	24
Figure 4.1	Solving time for different ways of using preprocessing and initialization.	62
Figure 4.2	Accuracy gain over CART for different ways of using preprocessing and initialization.	63
Figure 4.3	Number of columns added with original and merged SPs.	64
Figure 4.4	Accuracy gain over CART on training datasets with original and merged SPs.	64
Figure 4.5	Solving time for different ways of using constraints (4.1c).	65
Figure 4.6	MP value for different ways of using constraints (4.1c).	66
Figure 4.7	Training accuracy gain over CART for different ways of using constraints (4.1c).	67
Figure 4.8	Number of instances solved to optimality for different ways of using constraints (4.1c).	68
Figure 5.1	Interaction between different classes in the workflow of the CG Framework.	78

LIST OF SYMBOLS AND ACRONYMS

CG	Column Generation
ML	Machine Learning
MP	Master Problem
RMP	Restricted Master Problem
LP	Linear Program
MIP	Mixed Integer Linear Program
RL	Reinforcement Learning
NOTAM	Notice To Airmen
CART	Classification and Regression Trees

LIST OF APPENDICES

Appendix A Detailed Results for progressively strengthening and tuning MIP solvers for reoptimization 115

CHAPTER 1 INTRODUCTION

Machine learning (ML) methods aim to predict outcomes or uncover patterns from data, enabling various applications ranging from image recognition to financial forecasting. However, the interpretability and precision of these predictions can vary, often due to the heuristic nature of many ML algorithms. While the primary aim remains prediction, achieving accurate and interpretable results necessitates training these models on data.

Training an ML model entails optimizing parameters to minimize a predefined loss function or error metric. This optimization process can be conceptualized as solving an optimization problem where the objective is to find the model parameters that best fit the training data. Depending on the specific ML model employed, this optimization task may vary in complexity, with some models requiring the solution of highly nonlinear and high-dimensional optimization problems.

Furthermore, the interpretability of ML predictions varies depending on the model architecture and complexity. On the one hand, linear models such as linear regression or logistic regression offer straightforward interpretations, as the relationship between input features and output predictions is explicitly expressed through model coefficients. On the other hand, complex models like deep neural networks often lack interpretability due to their black-box nature, making it challenging to understand how input features contribute to the final prediction.

To address these challenges and enhance the interpretability and precision of ML predictions, integrating methodologies from operations research into the ML domain is crucial. One such methodology gaining traction is column generation (CG), which allows for the construction and solution of models with a large number of variables. Originally developed for large-scale optimization problems in operations research, CG has found application in fields such as transportation and logistics optimization [2], telecommunications network design [3], healthcare resource allocation [4], financial portfolio optimization [5], and energy systems optimization [6]. In this thesis, we explore the use of CG-based classifiers in ML and address practical challenges associated with applying them in practice.

This chapter is organized as follows. Section 1.1 covers the necessary background on CG and ML. Section 1.2 describes the research objectives of the thesis. Finally, Section 1.3 presents the thesis outline.

1.1 Background

This section presents the necessary background needed to comprehend the thesis. Section 1.1.1 describes CG and branch-and-price, and Section 1.1.2 gives a brief overview of ML.

1.1.1 Column Generation

The column generation technique [7] is used for solving linear programs (LPs) with a very large number of variables (columns). In this context, the LP is called the master problem (MP), which can be written as follows:

$$z^*_{MP} = \min_x \quad \sum_{i\in\mathcal{I}} c_i x_i \tag{1.1}$$

$$s.t. \quad \sum_{i\in\mathcal{I}} a_i x_i \geq b \tag{1.2}$$

$$x_i \geq 0, \forall i \in \mathcal{I} \tag{1.3}$$

where $c_i \in \mathbb{R}, a_i, b \in \mathbb{R}^m, \forall i \in \mathcal{I} = \{1, 2, ..., n\}$. Here, we assume that the number of constraints m is much smaller than the number of variables n, i.e., $m \ll n$.

Initially, a restricted version of the master problem (RMP) is created with a subset of variables $\mathcal{I}' \subseteq \mathcal{I}$. This RMP is then solved to optimality, and the dual costs π for constraints (1.2) are computed. Using these dual costs, the algorithm tries to find a column with a negative reduced cost. The reduced cost of a column i is given by $c_i - \pi^T a_i$. The following subproblem (sometimes referred to as the pricing problem) is solved to find a column with a negative reduced cost:

$$\bar{c}(\pi) = \min_{i\in\mathcal{I}} \quad c_i - \sum_{j=1}^{m} \pi_i a_i. \tag{1.4}$$

If the optimal cost $\bar{c}(\pi) \geq 0$, then the RMP solution is optimal for MP, as there are no more negative reduced cost columns (i.e., no violated constraints in the dual). Otherwise, the subproblem finds a column with a negative reduced cost that is added to the RMP, and the RMP is solved again. This continues until the optimality is reached or a limit on the number of iterations or on the time is reached. Notice that this algorithm is guaranteed to terminate since the number of variables (however large) is finite, and, in each iteration, we add a new column to the RMP. Note also that, to speed up convergence, multiple columns

can be added at each iteration when the algorithm used to solve the subproblem allows it.

The problem of interest is often modeled as a mixed integer linear program (MIP). In this case, the CG algorithm described above is only used for solving the LP relaxation of the original problem. When the optimal solution to the LP relaxation does not satisfy the integrality constraints, we solve the problem by branching in the MP. Note that this branching might not be performed directly on the individual variables [8]. Each node in the branching tree is solved using CG, and the entire process is called branch-and-price [9] or branch-price-and-cut when cutting planes are added to strengthen the linear relaxations.

Solving the complete mixed integer program to optimality can take a lot of time in some applications, as the LP relaxation at each node is solved using CG. In many such applications, we use a heuristic, and we only solve the root node of the integer MP using CG. Once the root node LP is solved, we solve the mixed integer program using the generated columns without generating new columns at each node. This heuristic is known as the restricted master heuristic [10].

1.1.2 Machine Learning

Machine learning belongs to the field of artificial intelligence. The main objective of this field is to develop algorithms that allow computers to solve complex tasks by learning from observations or experiences instead of explicitly being programmed.

ML algorithms are classified into three main categories: Supervised learning, Unsupervised learning, and Reinforcement learning. Although this thesis primarily focuses on supervised learning, we briefly introduce all categories below. The redears are invited to consult [11–13] for a detailed reference.

In the supervised learning problems, a set of data $\mathcal{D}$ is used for training a model (classifier) that will be used to make predictions for unseen data. The data contains the training features and the corresponding target label or value. The problem is called classification if the targets are categorical and regression if the targets are numerical values. The main objective is to predict the target label or value for an example outside the training data. This is done by learning a mapping from inputs to the targets such that the targets of the unseen examples can be approximated as well as possible. The approximations can be evaluated and optimized through error functions like least squared error or cross-entropy error.

The goal is to learn an approximate function that generalizes well to the unseen data. Consider a complex model (e.g., a high-degree polynomial function trying to approximate a linear function with minor noise) with many parameters. Such a model might be able to fit the

training data perfectly by minimizing the error function but may not do so well on unseen data. Moreover, the learned function can be highly dependent on the training data. Such models are said to have a high variance.

On the other hand, simple models like linear functions or decision trees with low depths can generalize well to the unseen data, but they have less representation power. They often fail to approximate a very complex function well enough. Such models are said to have a high bias. One typically has to deal with bias and variance trade-offs to find a suitable model for a specific task.

To address such issues, ML models often have hyperparameters that control the bias and variance of the model. Note that they are different from the parameters of the model. For instance, in decision trees (see, Section 2.2), the depth of the tree is considered a hyperparameter that influences the model's performance. We split the data into three sets: Training, Validation, and Test. Training data is used to learn the parameters. Validation data is used for testing the generalization for fixed values of hyperparameters and using them to tune the hyperparameters. Finally, test data is used to measure the generalization of the model.

In unsupervised learning problems, there is no target to predict. The main goal is to learn information about the data by analyzing the training data. For example, principal component analysis (PCA) takes high dimensional input data. It picks a fixed number of directions (k) on which to project the input data to reduce the dimension of the data to k without losing much information about the data. Similarly, clustering algorithms such as K-means help find underlying categories in the input data by detecting clusters. We do not discuss the unsupervised algorithms further in this thesis. The readers are referred to chapter 14 of [14] for a more detailed description of the unsupervised learning algorithms.

The reinforcement learning algorithms involve one or more agents interacting with the environment using a set of actions to maximize the expected reward. Unlike supervised and unsupervised learning, there is usually no training data. The agent learns to take the best actions for a given state by experimenting and observing the collected rewards—for example, an agent learning to play chess by playing against itself or other agents. The readers are referred to [15] for more details on reinforcement learning.

1.2 Research Objectives

A notable development in using LP/MIP based classifiers is the increasing utilization of the CG technique, particularly in creating models with a high number of variables. This technique has found application in diverse fields due to its ability to handle large-scale optimization

problems effectively. In the context of ML, researchers have started leveraging CG for its potential to enhance classifier performance and improve interpretability in ML (see, e.g., [16], [1]).

The application of CG techniques in practical ML contexts is met with formidable challenges. These encompass the management of large-scale training datasets, which often yield expensive CG models posing computational complexities. Additionally, imbalanced class distributions within these datasets introduce bias into classifier models. Furthermore, the strategic selection of initial columns in CG proves to be a critical yet intricate task. Finally, expediting the solving of subproblems to generate columns faster is essential for improving computational efficiency in CG-based methods. This thesis aims to explore the use of CG-based classifiers in ML and address these practical challenges.

Through these research objectives, the thesis seeks to advance the current understanding of the intersection between operations research and ML, specifically emphasizing enhancing the precision and applicability of ML methods through the integration of CG techniques.

The contributions of this thesis are outlined below, highlighting the novel elements introduced to the scientific literature:

1. Extension of CG-based classifiers to real-world applications, particularly in the classification of NOTAMs (NOtice To AirMen, messages containing important flight route related information), enhancing applicability.

2. Improvement of CG-based heuristic for learning classification trees, addressing computational challenges and reducing training times.

3. Development of a software framework facilitating the design and implementation of CG-based ML models, promoting accessibility and usability.

4. Innovation in addressing computational bottlenecks through novel approaches leveraging similarities among MIP subproblems, accelerating the solving process and enhancing efficiency.

These contributions collectively advance the understanding of the intersection between operations research and machine learning, particularly emphasizing the integration of column generation techniques to enhance the precision and applicability of ML methods in practical domains.

1.3 Thesis Outline

In Chapter 2, we present a literature review on various CG-based classifiers in ML and list some common observations. In Chapter 3, we extend the CG-based method for generating boolean decision rules to a real-world application for classifying NOTAMs that involves multiclass classification using one-vs-rest classification. We present the techniques we used to tackle the issues related to one-vs-rest classification, such as multiple outputs and class imbalances. This required changing the objective function of the MP as well as designing candidate weights to break ties for multiple outputs. In this work, most of the computation time is spent on solving the subproblems. We present a heuristic using a CP-SAT solver to generate columns faster and describe a way to reduce the search space in the subproblem. This work is published in [17].

In Chapter 4, we improve the CG-based heuristic in [1] for learning classification trees. First, we modify the subproblem model to reduce the number of subproblems in multiclass classification instances significantly. Next, we show that the data-dependent constraints in the master problem are implied and use them as cutting planes. Furthermore, we describe a separation model to generate data points for which the LP relaxation solution violates their corresponding constraints. Finally, we introduce a preprocessing and initialization routine that reduces the size of the master and subproblems, which in turn reduces the training times. This work is published in [18].

In Chapter 5, we describe a software framework that makes it easier for developers to design and implement CG-based ML models on top of scikit-learn [19]. The main goal is to provide an efficient, flexible, well-documented, open-sourced implementation of the common components used across various CG-based classifiers.

In the CG-based classifiers described in Chapters 3 and 4, we observed a notable challenge: a considerable portion of the training process is consumed by resolving subproblems, which emerges as a bottleneck. These subproblems, comprising MIPs that vary only in their objective function between CG iterations, are solved sequentially from scratch. However, in Chapter 6, we propose a novel approach to leverage the similarities inherent in these MIPs. Our method involves reusing information obtained from solving previous instances. Specifically, we detail techniques for reusing primal solutions and branching history, as well as strategies to enhance MIP solvers by incorporating concepts from reliability branching. Moreover, we introduce automated parameter optimization and the deactivation of inefficacious components within the MIP solver to expedite the solving process.

Finally, in Chapter 7, we summarize our work and discuss future research directions.

CHAPTER 2 LITERATURE REVIEW

In this chapter, we will review previous works where CG has been used in ML tasks. Section 2.1 describes a CG-based approach for learning boolean decision rules by [16]. Section 2.2 presents a CG-based heuristic for learning decision trees by [1]. Section 2.3 describes a CG-based approach for learning support vector machines by [20]. Section 2.4 presents a CG-based algorithm for learning ensemble classifiers using boosting by [21]. Finally, in Section 2.5, we conclude by listing some common observations in the listed CG-based models for ML.

Apart from the classifiers discussed in this chapter, CG is also used in computer vision applications (see e.g., [22]). This involves solving minimum weight set packing problems for data association using CG. These models are used for multi-person tracking, multi-person pose estimation, and multi-cell segmentation. Furthermore, recently CG is also used for generating a set of counterfactual explanations for a trained classifier (neural network or logistic regression) [23].

Boolean decision rules and decision trees described in Sections 2.1 and 2.2 respectively are interpretable classifiers. In recent years, the pursuit of interpretability in machine learning (ML) models has garnered significant attention, especially in the realm of classification tasks. Apart from the two classifiers discussed in this chapter, methodologies such as Local Interpretable Model-agnostic Explanations (LIME) [24] and Shapley values [25] have emerged as prominent tools in this endeavor. LIME offers a local explanation approach by generating interpretable models around specific instances, aiding in understanding complex model predictions. On the other hand, Shapley values, rooted in cooperative game theory, provide a global perspective by quantifying the contribution of each feature to the model's output, thus enhancing transparency and trust in classification outcomes. Since explainability is not the focus of this thesis, we do not explore it further. The readers are referred to [26] for more details on the topic.

2.1 Binary Classification using Boolean Decision Rules

The authors of [16] described a simple way to model the binary classification problem as a mixed integer program that helps improve the interpretability of the decision. This approach is a binary classification method that assumes that the inputs have binary features. The output is a boolean decision rule that evaluates to TRUE if the input is classified as positive and FALSE otherwise. The boolean decision rule is a Disjunctive Normal Form (DNF), i.e.,

OR-of-ANDs formula. A DNF formula is a set of clauses. If any of the clauses evaluates to TRUE, the formula evaluates to TRUE. Each clause contains a set of binary features and is set to TRUE if all the features are TRUE in the clause. An example of a DNF rule with two clauses is "(x_1 AND x_2) OR (x_3 AND x_5 AND x_7)", where x_i, $i = 1, 2, 3, 5, 7$, denote a binary feature. These binary rules also serve as explanations for the output class.

Some of the alternatives for learning boolean decision rules are as follows. The AQ method [27] attempts to learn the rule by considering one clause at a time heuristically. The clauses only cover the positive examples. The AQ method does not work if the dataset has noise (examples with the same features belonging to different classes). The CN2 method [28, 29] extends the AQ algorithm and generates clauses that may cover a limited number of negative examples. It generates the clause leveraging ideas from the ID3 algorithm [30]. More precisely, ID3 is used for selecting decision tests in a decision tree node. The ID3 algorithm measures the impurity of a node in the decision tree by an entropy function. The decision test that minimizes the impurity is selected for each node. Similarly, the CN2 algorithm computes the impurity of the clauses (using the entropy function) to evaluate their quality. The clauses that cover mostly the positive examples and a few negative examples are less impure. Such clauses are added to the decision rules. This allows the CN2 algorithm to work with the data with noise. The RIPPER algorithm [31] learns the rules using a separate-and-conquer strategy. It starts by treating the least common class in the dataset. It generates the clauses by using a general-to-specific method for growing clauses. Then, it prunes the generated clauses depending on their performance in the validation set. These algorithms primarily aim for accuracy and do not focus on reducing the rule complexity. Hamming clustering [32] is another way of learning boolean rules. Here, the rules are learned by clustering the inputs and then pruning rules to generalize. A detailed survey for rule learning is given in [33].

In [16], the authors describe a simple MIP model to construct a DNF boolean decision rule. In this model, they use binary variables to represent clauses. The decision rule includes the clause if the corresponding binary variable takes value 1. We also want our decision rule to be interpretable. A decision rule containing too many or too large clauses is less interpretable. To avoid generating such rules, they use a hyperparameter C to limit the complexity of the decision rule. The complete model is based on the notation in Table 2.1.

Table 2.1 Notations for the MIP (2.1).

Sets	
$\mathcal{P}$	Set of all positive examples in the dataset.
$\mathcal{Z}$	Set of all negative examples in the dataset.
$\mathcal{K}$	Set of all clauses.
$\mathcal{K}_i$	Set of all clauses satisfied by example $i \in \mathcal{P} \cup \mathcal{Z}$.
Parameters	
c_k	Complexity of clause $k \in \mathcal{K}$. Typically, it is set to 1 plus the length of the clause.
C	Complexity of the explanation.
Decision Variables	
w_k	Binary variable indicating if clause $k \in \mathcal{K}$ is selected.
ξ_i	Binary variable indicating if example $i \in \mathcal{P}$ does not satisfy any selected clauses.

$$z_{MIP} = min \quad \sum_{i \in \mathcal{P}} \xi_i + \sum_{i \in \mathcal{Z}} \sum_{k \in \mathcal{K}_i} w_k \tag{2.1a}$$

$$s.t. \quad \xi_i + \sum_{k \in \mathcal{K}_i} w_k \geq 1, i \in \mathcal{P} \tag{2.1b}$$

$$\sum_{k \in \mathcal{K}} c_k w_k \leq C \tag{2.1c}$$

$$\xi_i \geq 0, i \in \mathcal{P} \tag{2.1d}$$

$$w_k \in \{0, 1\}, k \in \mathcal{K} \tag{2.1e}$$

The first sum in the objective is the penalty for the misclassified positive examples, and the second is the penalty for the misclassified negative examples. Constraints (2.1b) ensure that each positive example satisfies at least one of the selected clauses or the associated penalty variable is set to 1. Constraints (2.1c) control the overall complexity of the generated DNF formula. The number of clauses is exponential, so the columns for w_k are generated by CG through a subproblem.

The reduced cost for a column w_k is given by $\sum_{i \in \mathcal{Z}} \delta_i - \sum_{i \in \mathcal{P}} \mu_i \delta_i - \lambda c_k$, where $\mu_i \geq 0$ are the dual costs for constraints (2.1b) and $\lambda \leq 0$ is the dual cost for constraint (2.1c). The binary variable δ_i is set to 1 if the ith example satisfies the clause w_k.

Using the notation provided in Table 2.2, the SP used to generate clause variables w_k can be formulated as follows for a given RMP optimal dual solution (λ^*, μ^*):

Table 2.2 Notations for the MIP (2.2).

Sets	
J	Set of all features.
$S_i = \{j \in J : x_{ij} = 0\}$	The zero-valued features in sample $i \in \mathcal{P} \cup \mathcal{Z}$.
Parameters	
D	Maximum size of the generated clause.
Decision Variables	
z_j	Binary variable indicating if feature j is present in the generated clause.
δ_i	Binary variable indicating if example i is satisfied by the generated clause.

$$z_{CG} = \min \quad \sum_{i \in \mathcal{Z}} \delta_i - \lambda^* \left(1 + \sum_{j \in J} z_j\right) - \sum_{i \in \mathcal{P}} \mu_i^* \delta_i \tag{2.2a}$$

$$s.t. \quad \delta_i + z_j \leq 1, j \in S_i, i \in \mathcal{P} \tag{2.2b}$$

$$\delta_i \geq 1 - \sum_{j \in S_i} z_j, i \in \mathcal{Z} \tag{2.2c}$$

$$\sum_{j \in J} z_j \leq D \tag{2.2d}$$

$$z_j \in \{0, 1\}, j \in J \tag{2.2e}$$

$$\delta_i \geq 0, i \in \mathcal{Z} \tag{2.2f}$$

In the objective (2.2a), the term $1 + \sum_{j \in J} z_j$ is equal to the generated clause complexity, i.e., to 1 plus the clause size. This objective function corresponds to the reduced cost of the clause generated. Negative objective values identify clauses with a negative reduced cost, and the clause can be added to the RMP in (2.1). The clause is constructed by selecting the features for which the corresponding z variables are set to 1. The constraints (2.2b) - (2.2c) relate the selected features with the δ variables. Constraint (2.2d) controls the complexity of the generated clause. Note that the variables δ_i are implied binary, so they need not be encoded as binary.

The model (2.1) outputs a DNF formula at the end of the training (solving the integer MP in this case). The DNF formula contains the clauses for which the w variables are set to 1. This DNF formula is used for classifying new unseen inputs. If an input is classified as positive,

the clauses that are satisfied by that example in the DNF formula serve as an explanation for "why the input was classified as positive".

The number of constraints and clause variables in (2.1) is of $O(|\mathcal{P}|)$. The subproblem (2.2) uses many more constraints. The number is of $O(|\mathcal{P}||\mathcal{K}| + |\mathcal{Z}|)$. The number of variables in (2.2) is of $O(n)$, where n is the total number of input data points.

The authors binarize the features when the features are not binary. They use sample deciles for converting a non-binary feature into a set of binary features. They also use negations of the binary features and include them in the set of all features.

In [16], the columns are only generated for the root node until the decided time limit is reached. Once the process hits the time limit, the RMP with its generated columns is directly passed to a MIP solver. This is known as the restricted master heuristic [10]. Because of this, the problem is sometimes not solved to optimality. The CG approach faces computational difficulties for medium and large datasets (more than a few thousand input data points with a few hundred or more binary features). In those cases, heuristics are used, like selecting a subset of constraints and columns while solving the subproblem or heuristically generating columns instead of solving (2.2) as a MIP. In the paper, the authors have not described how the initial columns are selected in the MP.

The authors finally compare their approach with existing state-of-the-art approaches on various datasets. They compare with Bayesian Rule Sets (BRS) [34], the alternating minimization (AM) and block coordinate descent (BCD) algorithms [35], WEKA [36], RIPPER [31], decision tree learner CART [37], random forests [38], and various other techniques from ML. This CG approach achieves comparable accuracy to those algorithms and provides an explanation that is easy to interpret. Recently, the authors extended this method in the fairness setting in [39].

2.2 Decision Trees

The decision trees are a well-known model for interpretable AI for classification tasks. The goal here is to construct a tree-like structure such that each non-leaf node evaluates the input data point on a fixed query and passes it to the left or right child, depending on whether the input satisfies the query. Finally, each leaf node is assigned a class label. When the input reaches a leaf node, the assigned class for that leaf node is the output prediction. In the learning phase, the training data is divided into a tree-like structure, with the root node representing the entire dataset. At each non-leaf node, the data is split into two or more branches. The splitting criteria can be based on one or more features.

The standard ML algorithms for training decision trees, like CART [37] and ID3 [30], use heuristics that optimize for one-depth level accuracies. They are fast in terms of training times. However, the generated trees are suboptimal in terms of overall accuracy.

Learning optimal binary decision trees is an NP-complete problem [40]. Many authors have created optimization models for learning decision trees optimal in terms of accuracy. [41], [42], [43], [44], and [45] introduced MIPs for training decision trees. For a recent survey on continuous and MIP models for training decision trees, refer to [46]. [47] described a mixed integer quadratic model for learning optimal trees. [48, 49] introduced mixed integer non-linear models for training decision trees. [50] proposed a continuous optimization approach for training randomized decision trees. [51] introduced a dynamic-programming-based exact algorithm to compute optimal decision trees. [52] presented a SAT model to train decision trees. Finally, [53] proposed a constraint programming model for training decision trees for binary classification tasks.

These models can learn trees with better accuracy compared to CART and ID3. Sometimes, they can find the optimal decision tree for a given dataset. However, the biggest issue with these models is scalability. Except for the binary classification model of [53], no other paper (from the ones cited above) showed results on datasets with more than 10,000 rows.

Firat et al. [1] present a CG-based model for generating decision trees. The authors focus on creating univariate binary decision trees of a fixed depth. The trees can handle multi-class classification, unlike in [41]. However, the authors do not claim to generate optimal classification trees. Instead, they try to address the scalability issues in the other models.

The model uses decision variables for each path from the root to the leaf. There are exponentially many paths for a fixed set of split decisions at each node. These paths are generated using CG. The MP ensures that the selected paths are consistent with each other (forms a tree), and the subproblem ensures finding a path with an optimal reduced cost. This includes ensuring the training data is correctly following/not following the paths.

The integer MP model is given as below:

Table 2.3 Notation for the MIP (2.3).

Sets	
R	Set of rows in the dataset.
F	Set of features in the dataset.
N_{lf}, N_{int}	Sets of leaf and internal nodes in the decision tree.
$p_{BT}(l)$	Set of nodes in the paths to leaf l in binary tree.
DP_l	Set of decision paths ending in leaf l.
$R^l(p)$	Subset of rows directed to leaf l through path p.
S_j	Set of candidate split checks for node j.
Parameters	
$s_p(j)$	Split check assigned at node j in path p.
$CP(p)$	Number of correct predictions for path p.
Decision Variables	
x_p	Binary variable indicating if path $p \in DP_l$ is assigned to leaf $l \in N_{lf}$.
$\rho_{j,a}$	Binary variable indicating if split check $a \in S_j$ is assigned to node $j \in N_{int}$.

$$Max \quad \sum_{l \in N_{lf}} \sum_{p \in DP_l} CP(p) x_p \tag{2.3a}$$

$$s.t. \quad \sum_{p \in DP_l} x_p = 1, \quad \forall l \in N_{lf} \tag{2.3b}$$

$$\sum_{l \in N_{lf}} \sum_{\substack{p \in DP_l: \\ r \in R^l(p)}} x_p = 1, \quad \forall r \in R \tag{2.3c}$$

$$\sum_{\substack{p \in DP_l: \\ s_p(j) = a}} x_p = \rho_{j,a}, \quad \forall l \in N_{lf}, j \in p_{BT}(l) \cap N_{int}, a \in S_j \tag{2.3d}$$

$$x_p \in \{0,1\}, \quad \forall p \in DP_l, l \in N_{lf} \tag{2.3e}$$

$$\rho_{j,a} \in \{0,1\}, \quad \forall j \in N_{int}, a \in S_j. \tag{2.3f}$$

The objective directly minimizes the misclassification error. Constraints (2.3b) ensure that exactly one path is selected for each leaf. Constraints (2.3c) ensure that each data point follows exactly one selected path. Finally, constraints (2.3d) ensure that the selected paths are consistent with each other in terms of splitting criteria on common nodes. Note that in this model, the variable upper bounds are implied and can, hence, be dropped while solving

the LP relaxation.

Relying on the additional notation presented in Table 2.4, the SP for leaf $l \in N_{lf}$ and target $t \in T$ can be formulated as the following MIP:

$$Max \quad \sum_{r \in R_t} y_r - \alpha_l - \sum_{j \in p_{BT}(l)} \sum_{a \in S_j} \gamma_{l,j,a} u_{j,a} - \sum_{r \in R} \beta_r y_r \tag{2.4a}$$

$$s.t. \quad \sum_{a \in S_j} u_{j,a} = 1, \quad \forall j \in p_{BT}(l) \tag{2.4b}$$

$$y_r \leq \sum_{a \in S_j \cap T(r)} u_{j,a}, \quad \forall j \in LC(l), r \in R \tag{2.4c}$$

$$y_r \leq \sum_{a \in S_j \cap F(r)} u_{j,a}, \quad \forall j \in RC(l), r \in R \tag{2.4d}$$

$$\sum_{j \in LC(l)} \sum_{a \in S_j \cap T(r)} u_{j,a} + \sum_{j \in RC(l)} \sum_{a \in S_j \cap F(r)} u_{j,a} - (k-1) \leq y_r, \quad \forall r \in R \tag{2.4e}$$

$$\sum_{j \in p_{BT}(l)} u_{j,a} \leq 1, \quad \forall a \in \bigcup_{j \in p_{BT}(l)} S_j \tag{2.4f}$$

$$y_r \in \{0,1\}, \quad \forall r \in R \tag{2.4g}$$

$$u_{j,a} \in \{0,1\}, \quad \forall j \in RC(l) \cup LC(l), a \in S_j. \tag{2.4h}$$

The objective is the reduced cost of the path being generated. Constraints (2.4b) ensure that only one split is selected for each node in the path. Constraints (2.4c)-(2.4e) evaluate if the row r would reach the leaf by following the path. Constraints (2.4f) ensure that each split is selected only once in a path.

Since the subproblem is solved independently for each leaf and target value, a path with the wrong target might be generated because of the positive reduced cost. In that case, a post-processing step is performed to correct the target label. The authors also use randomly generated paths as a heuristic to generate columns that is used before solving the pricing subproblem. The restricted split checks for each node are generated by training a small amount of training data with CART multiple times and selecting the checks that appear more often.

The initial columns are selected by running the CART algorithm on the complete input data and using the paths generated in the solution tree. The authors mention that they do not use a branch-and-price algorithm for solving the integer MP. Instead, they use the restricted master heuristic. The authors mention that they experimented with branch-and-price but faced difficulty choosing branching rules. They mention developing the complete branch-and-

Table 2.4 Additional notation for the SP (2.4) defined for leaf l and target t.

Sets	
R_t	Set of rows in the dataset with target t.
$LC(l)$	Set of nodes in N_{int} that have left child in $p_{BT}(l)$.
$RC(l)$	Set of nodes in N_{int} that have right child in $p_{BT}(l)$.
$T(r)$	Set of split checks for which row r takes the left branch: $\{a = (f_a, \mu_a) \in \cup_{j \in N_{int}} S_j : v_r^{f_a} \leq \mu_a\}$.
$F(r)$	Set of split checks for which row r takes the right branch: $\{a = (f_a, \mu_a) \in \cup_{j \in N_{int}} S_j : v_r^{f_a} > \mu_a\}$.
Parameters	
k	Depth of the decision tree, levels are indexed by $h = 0, ..., k-1$.
v_r^f	Value of feature f in row r.
α_l	Dual value of constraint (4.1b) for leaf l.
β_r	Dual value of constraint (4.1c) for row r.
$\gamma_{l,j,a}$	Dual value of constraint (4.1d) for leaf l, node j, and split check a.
Decision Variables	
y_r	Binary variable indicating if row $r \in R$ reaches leaf l.
$u_{j,a}$	Binary variable indicating if split check $a \in S_j$ is assigned to node $j \in p_{BT}(l)$.

price algorithm as future research work. The authors also provide the integrality of the MP solutions as a measure of how good the relaxation is. For smaller depths (2-3), they observe that a large fraction of the solutions are integer.

The number of variables that are not generated by CG in the MP is of $O(2^D)$ and the number of constraints is of $O(2^D + n)$. Each subproblem has $O(2^D + n)$ variables and $O(nD)$ constraints. This is smaller than the number of constraints in the OCT model. However, this causes practical limits for the tree depth values for large data.

The authors also describe the variations of this model to focus on minimizing the false negatives, obtaining trees with a low number of leaves, and incorporating constraints on the minimum number of rows to end in a leaf.

As per the results presented in the paper, this approach also performs slightly better than CART on large data. For small data with low-depth trees, this approach is comparable to the OCT model. Note that the OCT model is not capable of handling large input data.

2.3 Linear Support Vector Machines

The support vector machine (SVM) [54] is an algorithm used in ML for binary classification. In this algorithm, we try to find a linear decision boundary that separates the two classes.

When the two classes are linearly separable, there might be more than one decision boundary separating the two classes. In SVM, we try to find the boundary that maximizes the margin (distance to the closest points) that separates the two classes.

Let the input data be denoted by $x \in \mathbb{R}^{n \times p}$ and the class labels be denoted by $y \in \{-1, 1\}^n$. The linear decision boundary is represented using parameters β, β_0 as $x^T\beta + \beta_0$. When the input x_i is correctly classified, we have $y_i(x_i^T\beta + \beta_0) > 0$. We assume the margins are a unit distance from the decision boundary. This is a valid assumption because the distance minimization problem is not affected by the scaling of β, β_0. Hence, all the correctly classified data points will satisfy $y_i(x_i^T\beta + \beta_0) \geq 1$.

However, the data is not always linearly separable. In such cases, we relax those constraints and add a penalty ξ_i when the data point x_i lies on the wrong side of the margin. The constraint is given by $\xi_i + y_i(x_i^T\beta + \beta_0) \geq 1$. The interpretation of the ξ values is shown in Table 2.5. The ξ_i values also represent the hinge loss function $\Big(1 - y_i(x_i^T\beta + \beta_0)\Big)_+$ where $(a)_+ := max\{a, 0\}$. This error function is also called soft margin error.

Table 2.5 Interpretation of ξ values

	Classification	Place of data point
$\xi = 0$	Correct	On the correct side of the margin
$0 < \xi \leq 1$	Correct	Between boundary and the margin
$\xi > 1$	Incorrect	Wrong side of the boundary

Hence, the SVM objective is to minimize the sum of the ξ values. A regularization term is added to the objective to ensure that the coefficients β are not too large. This helps with controlling the overfitting problem.

In [20], the authors describe a LP model to compute the SVM parameters. They model the penalized L1 SVM problem described as follows:

$$\underset{\beta \in \mathbb{R}^p, \beta_0 \in \mathbb{R}}{min} \sum_{i=1}^{n} \Big(1 - y_i(x_i^T\beta + \beta_0)\Big)_+ + \lambda||\beta||_1 \tag{2.5}$$

Here, λ is the regularization parameter. Note that L1 regularization leads to a sparser β. A similar approach using CG is also described for computing estimators of Dantzig selectors in [55]. The paper [56] talks about the L2 SVM problem where the regularization is done using the L2 norm.

The problem (2.5) can be written as a LP. The authors in [20] find that commercial LP

solvers like Gurobi and Cplex work very well for small- and medium-sized problems but find it hard to solve when n and/or p is large (around ten thousand). For small-sized problems where $n \approx 100$ and $p \approx 10^4$, the commercial solver solves the problem faster than the first order methods based on the Alternating Direction Method of Multipliers (ADMM) [57] with Nesterov's smoothing [58]. The authors in [20] proposed using column and constraint generation for solving large-scale SVM instances.

Let $\alpha_i = 1 - y_i(x_i^T\beta + \beta_0)$. We have $\xi_i = (\alpha_i)_+$. The sparsity in α_i can be achieved using CG, and sparsity in β can be achieved using constraint generation. The authors describe the complete L1 SVM LP model as follows:

$$\min_{\xi\in\mathbb{R}^n,\beta_0\in\mathbb{R},\beta^+,\beta^-\in\mathbb{R}^p} \quad \sum_{i=1}^{n}\xi_i + \lambda\sum_{j=1}^{p}\beta_j^+ + \lambda\sum_{j=1}^{p}\beta_j^- \tag{2.6a}$$

$$s.t. \quad \xi_i + y_i x_i^T\beta^+ - y_i x_i^T\beta^- + y_i\beta_0 \geq 1, \forall i \in \{1,...,n\} \tag{2.6b}$$

$$\xi \geq 0, \beta^+ \geq 0, \beta^- \geq 0 \tag{2.6c}$$

Here, $\beta = \beta^+ - \beta^-$. We split β to model the L1 norm. Because of the objective direction, only one of the β^+, β^- can take a positive value in the optimal solution.

Let π_i be the dual variables associated with constraints (2.6b), then the dual problem is described as follows:

$$\max_{\pi\in\mathbb{R}^n} \quad \sum_{i=1}^{n}\pi_i \tag{2.7a}$$

$$s.t. \quad -\lambda \leq \sum_{i=1}^{n} y_i x_{ij}\pi_i \leq \lambda, \forall j \in \{1,...,p\} \tag{2.7b}$$

$$y^T\pi = 0 \tag{2.7c}$$

$$0 \leq \pi_i \leq 1, \forall i \in \{1,...,n\} \tag{2.7d}$$

The reduced costs for variables β^+, β^- are derived using the dual constraint, and the minimum of the two reduced costs for each pair is given by

$$min\{\bar{\beta}_j^+, \bar{\beta}_j^-\} = \lambda - \left|\sum_{i=1}^{n} y_i x_{ij}\pi_i^*\right| \tag{2.8}$$

where π_i^* represents the dual costs for constraint i in (2.6). These reduced costs are used for

the CG algorithm. Initially, the primal problem (2.6) is solved with a subset of columns $\mathcal{J}$, and the columns with negative reduced costs are generated as needed.

For the selection of the initial columns, the authors use various heuristics.

1. Correlation screening: They compute the absolute inner product of each column with the target values and then select the ones with the highest absolute inner product values.

2. Regularization path: They start with a higher value of λ to obtain initial solutions and then decrease the value of λ. The solution and the columns of the previous value of λ serve as a warm start.

3. First-order methods: They use Nesterov's smoothing method followed by proximal gradient descent to obtain a low-accuracy solution. This solution is used for computing the initial columns. They also use this to identify the initial rows when they apply constraint generation, as explained below.

When the training data is large, they use constraint generation to solve the LP. Initially (2.6) is solved using a subset of constraints $\mathcal{I}$, and the violated constraints (i.e., constraints corresponding to dual variables with positive reduced costs) are added as needed. The constraint generation is done using the reduced costs of π_i given by

$$\bar{\pi_i} = 1 - y_i(x_i^T \beta^* + \beta_0^*) \tag{2.9}$$

where β^*, β_0^* represent the optimal solution values of (2.6).

For generating initial rows, they also describe another heuristic called subsampling. They solve (2.6) for various subsamples of the input data and compute the average solution $\bar{\beta}$. They find the rows misclassified by $\bar{\beta}$ and use them as initial rows. Similarly, they use this solution to compute an initial set of columns. In the case of CG, they do not compute the solution of the LP and instead compute the columns using correlation screening.

The authors also combine column and constraint generation when both n and p are large. Finally, the authors describe similar models for some variants of SVM, like group SVM and slope SVM.

Note that we do not have an exponential number of columns or constraints in SVM. The number of columns is of $O(p)$ and the number of constraints is of $O(n)$. The authors used column and constraint generation because, often, the number of features and data points is

too large for an LP solver to handle. Also, note that the problem solved is an LP, not a MIP. Hence, in this case, the computational cost is lower than other ML models where CG is used to solve a MIP.

2.4 Boosting

In ML, we often train more than one model and combine their predictions to produce a better prediction. Boosting is one of the ways to create such an ensemble. In boosting algorithms, we create a set of weak learners that are good at predicting a particular segment of inputs, and then we compute weights for those weak learners to produce a combined prediction. Note that the learners we use are "weak" because they alone can not produce good predictions for the entire dataset. One such boosting method is AdaBoost [59].

In [21], the authors developed an LP-based boosting method that compares with AdaBoost. The authors begin with a binary classification task. The algorithm can be generalized to multiclass classification and regression as well. The training data is given as $\{(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)\}$. Let $y_i \in \{-1, 1\}$. Note that for a prediction $h(x) \in \{-1, 1\}$, we have $h(x)y > 0$ for correct classification and $h(x)y < 0$ for incorrect classification. Assume a set of weak learners $\{h_1, h_2, ..., h_n\}$ are available. Each of these weak learners h_j produce output $h_j(x_i) \in \{-1, 1\}$ for each of the input x_i. This creates a matrix H with m rows and n columns. Each column represents a weak learner. The algorithm tries to find weights a_j for each of the weak learners h_j such that the weighted linear combination of the predictions minimizes the soft margin error (also described in SVM Section 2.3).

The LP model is given by

$$\min_{a,\xi} \quad \sum_{j=1}^{n} a_j + C \sum_{i=1}^{m} \xi_i \tag{2.10a}$$

$$s.t. \quad y_i H_i a + \xi_i \geq 1, \forall i \in \{1, ..., m\} \tag{2.10b}$$

$$a_j \geq 0, \forall j \in \{1, ..., n\} \tag{2.10c}$$

$$\xi_i \geq 0, \forall i \in \{1, ..., m\} \tag{2.10d}$$

The objective is minimizing the soft margin error that is similar to SVM. The weights a are also minimized to avoid overfitting. Constraints (2.10b) ensure that the margin violations (variables ξ) are set correctly.

An alternative model with parameter D is given by

$$\max_{a,\xi,\rho} \quad \rho - D\sum_{i=1}^{m}\xi_i \tag{2.11a}$$

$$s.t. \quad y_i H_i a + \xi_i \geq \rho, \forall i \in \{1, ..., m\} \tag{2.11b}$$

$$\sum_{j=1}^{n} a_j = 1 \tag{2.11c}$$

$$a_j \geq 0, \forall j \in \{1, ..., n\} \tag{2.11d}$$

$$\xi_i \geq 0, \forall i \in \{1, ..., m\} \tag{2.11e}$$

Let u be the dual variables for constraints (2.11b) and β be the dual variable for constraint (2.11c). The dual of (2.11) is given by

$$\min_{u,\beta} \quad \beta \tag{2.12a}$$

$$s.t. \quad \sum_{i=1}^{m} u_i y_i H_{ij} \leq \beta, \forall j \in \{1, ..., n\} \tag{2.12b}$$

$$\sum_{i=1}^{m} u_i = 1 \tag{2.12c}$$

$$0 \leq u_i \leq D, \forall i \in \{1, ..., m\} \tag{2.12d}$$

The two models (2.10) and (2.11) are equivalent for appropriate choice of parameters D and C [60]. In other words, the primal and dual solution values for one model can be computed without solving the LP if the primal and dual solution values of the other model are known.

The dual variables u_i represent the misclassification cost for input x_i. The term $\sum_{i=1}^{m} u_i y_i H_{ij}$ represents the score for classifier h_j as the weighted sum of the correctly classified points minus the weighted sum of the incorrectly classified points.

In this LP, the number of possible weak learners is exponential in the size of the input data (2^m). Hence, the algorithm starts with equal weights $u_i = 1/m$ for all inputs and no weak learner. Given the weights, the algorithm assumes that there is a way to generate a weak learner h_j that violates the dual constraint and is then added to the LP as a generated column. Note that many algorithms in ML (for example, C4.5, CART) take weighted inputs and return a learner that minimizes the weighted misclassification error. These algorithms need not return a learner that violates the dual constraint the most. However, computing the learner that violates the dual constraint most (best reduced cost for primal) speeds up

convergence. After adding the generated column, the LP is solved again, and this process is repeated until no dual constraint is violated. Finally, the ensemble prediction is computed as the sign of $f = \sum_{j=1}^{n} a_j h_j$.

The algorithm is called LPBoost. The same formulation also works if the classifiers produce outputs in the confidence interval $[-1, 1]$ instead of $\{-1, 1\}$. The authors also describe a slight modification to generalize the algorithm for regression tasks.

Here, we again do not have an integer MP. The number of non-generated columns and constraints is of $O(m)$, where m is the size of the training dataset. The subproblem is not a LP and is solved by some ML oracle. The authors report that the time to compute the LP solution is much less than the time to generate weak learners (columns).

The authors used decision tree stumps (depth one decision trees) and C4.5 as base learners for their experiments. The LPBoost algorithm compares well with AdaBoost in terms of computational time and accuracy in the experiments. The authors mention the following advantages over gradient-based boosting (like AdaBoost): Finite termination at a globally optimal solution, well-defined convergence criteria, and fewer weak learners in the optimal ensemble.

2.5 Common Observations

In the ML algorithms with the use of CG described above, we observe the following:

1. Size of the MP: The MP requires an order of $O(n)$ constraints, where n is the size of the training data. This is, however, rarely the bottleneck as the subproblem is much larger and harder to solve, except for SVM where the authors in [20] used subsampling to select the initial set of rows and then used constraint generation.

2. Size of the subproblem: The subproblem for the boolean decision rule generation and the decision tree algorithm requires at least one constraint per input data. This becomes an issue when the training dataset is large. For SVM, the subproblem is much smaller and easier to solve (it does not have integer variables). The other two algorithms use a few heuristics to speed up the CG process. They do not describe any clever way of selecting a subset of rows to make the subproblem smaller. LPBoost also uses the entire dataset to generate new columns.

3. Complete optimization: For the model in SVM and LPBoost, branch-and-price is not needed as there is no integer MP. However, the MP is integer for the boolean decision rules and the decision tree algorithms. Nevertheless, since the subproblem is too large,

the authors in [16] and [1] use CG only for the root node and use the restricted master heuristic.

4. Initial columns: The authors in the SVM and the decision trees papers [20] and [1] use the algorithm they are comparing against to generate the initial set of columns. The SVM paper also uses other heuristics like correlation screening and regularization path that do not generalize for the other algorithms.

Although size of the MP is not a bottleneck, we introduce a cutting plane generation technique in Chapter 4 to control the MP size. To alleviate the large size of the SP, new heuristics for the SP are developed in Chapter 3 and some preprocessing is applied in Chapter 4. Furthermore, in Chapter 4, the SPs are merged to yield a much smaller number of them, without increasing too much the complexity of solving a single one. In our work, we do not explore complete optimization using branch-and-price. Finally, in Chapter 4, in addition to running CART once on the complete input data like in [1], we also run CART multiple times on randomly selected partial input data to generate more initial columns.

CHAPTER 3 ARTICLE 1: EXPLAINABLE PREDICTION OF QCODES FOR NOTAMS USING COLUMN-GENERATION

Authors: Krunal Kishor Patel, Guy Desaulniers, Andrea Lodi, and Freddy Lecue

Submitted on August 1, 2022 and published on February 22, 2023 in Journal of the Operational Research Society, Volume 75 Issue 1, pages: 173-183, DOI: 10.1080/01605682.2023.2181715

Abstract

A NOtice To AirMen (NOTAM) contains important flight route related information. To search and filter them, NOTAMs are grouped into categories called QCodes. In this paper, we develop a tool to predict, with some explanations, a Qcode for a NOTAM. We present a way to extend the interpretable binary classification using column generation proposed in [16] to a multiclass text classification method. We describe the techniques used to tackle the issues related to one-vs-rest classification, such as multiple outputs and class imbalances. Furthermore, we introduce some heuristics, including the use of a CP-SAT solver for the subproblems, to reduce the training time. Finally, we show that our approach compares favorably with state-of-the-art machine learning algorithms like Linear SVM and small neural networks while adding the needed interpretability component.

Keywords: Interpretable text classification; Boolean decision rules; Column generation; NOTAM; Multiclass text classification

3.1 Introduction

A NOtice To AirMen or NOtice To Air Missions (NOTAM) [61] is a notice sent to alert aircraft pilots of potential hazards along a flight route or at a location that could affect the flight. Generally, a NOTAM states the abnormal status of a component of the National Airspace System (NAS). NOTAMs concern the establishment, condition, or change of any facility, service, procedure, or hazard in the NAS. Sometimes a NOTAM is not alerting of potential hazard, but only providing some practical information that is not critical.

The NOTAMs are grouped by a label called Qcode [62]. The aircraft pilots use this Qcode to search and filter the NOTAMs. Each NOTAM is assigned one Qcode. The Qcode is a five letter label. The first letter is always 'Q'. The second and the third letter identify

the subject being reported. The fourth and fifth letters identify the status of operation of the subject being reported. These Qcodes are assigned manually to each message by the airport authorities writing NOTAMs. Since the Qcode categories are too many and partially redundant, people sometimes assign them wrongly, or do not assign any and choose "QXXXX". Our goal is to predict the Qcode for a NOTAM given its message content.

Standard machine learning algorithms can predict the Qcode for a given message. However, they lack explanation. Using boolean decision rules, we are able to tackle the explainability part. But, to achieve an accuracy similar to that obtained by standard machine learning algorithms, we have to predict a small list of Qcodes instead of a single Qcode for a given message. A human will then pick a final Qcode from this list (along with the explanations). We believe that this is better than picking a Qcode from a large pool of possible Qcodes or blindly trusting a machine learning model's output without any explanation.

Note that a NOTAM contains many fields other than just the message. However, the industrial partner Thales asked for a solution using only the message field to predict the Qcode.

Figure 3.1 shows a typical NOTAM example for London Heathrow Airport.

```
A1234/06 NOTAMR A1212/06
Q)EGTT/QMXLC/IV/NBO/A/000/999/5129N00028W005
A)EGLL
B)0609050500
C)0704300500
E)DUE WIP TWY B SOUTH CLSD BTN 'F' AND 'R'. TWY 'R' CLSD BTN 'A' AND 'B' AND
DIVERTED VIA NEW GREEN CL AND BLUE EDGE LGT. CTN ADZ
```

Figure 3.1 A typical NOTAM for London Heathrow Airport.

As shown, the NOTAM has many fields. We primarily focus on the Qcode (QMXLC) and the message (field E) that reads as follows:

> Due to work in progress (DUE WIP), taxiway "B South" is closed between "F" and "R" (TWY B SOUTH CLSD BTN 'F' AND 'R'). Taxiway "R" is closed between "A" and "B" (TWY 'R' CLSD BTN 'A' AND 'B') and is diverted via a new green centre line and blue edge lighting (AND DIVERTED VIA NEW GREEN CL AND BLUE EDGE LGT). Caution advised (CTN ADZ).

Explainability is a well-recognized goal for machine learning prediction tasks. The Qcodes can be predicted without explanation for the NOTAMs. However, since the Qcodes are assigned

by humans (the airport authorities) and used by humans (the pilots), the explanation for each Qcode increases confidence and ensure trust by airport authorities. This is what we are looking to achieve in this work.

To classify NOTAMs into Qcodes, we use an extension of the mixed-integer programming (MIP) model proposed in [16] and solved by column generation. This approach is a binary classification method that assumes that the inputs have binary features. The output is a boolean decision rule that evaluates to TRUE if the input is classified as positive and FALSE otherwise. More precisely, the boolean decision rule is a Disjunctive Normal Form (DNF), OR-of-ANDs formula. A DNF formula is a set of clauses. If any of the clause evaluates to TRUE, the formula evaluates to TRUE. Each clause contains a set of binary features and is set to TRUE if all the features are TRUE in the clause. An example of a DNF rule with two clauses is "(x_1 AND x_2) OR (x_3 AND x_5 AND x_7)", where x_i, $i = 1, 2, 3, 5, 7$, denote a binary feature. These binary rules also serve as explanations for the output class.

In order to extend the approach in [16] to our context, we had to face a few challenges. First, the NOTAM classification task is a multiclass classification problem. A one-vs-rest classification method is a natural way to extend a binary classification method to a multi-class classification context. In addition, the model proposed in [16] is designed for balanced datasets (similar number of positive and negative inputs in the training data). However, because we are using one-vs-rest classification, the number of negative data points is much larger compared to the number of positive data points. Another issue is that an input can satisfy the boolean decision rules of more than one class. This is a common issue with one-vs-rest methods. Finally, in our context, we could not use the entire training dataset for learning the boolean decision rule for a class because that leads to very large linear programs. In this paper, we describe how to effectively adapt and extend the method introduced in [16] to address all these challenges, and we show that the resulting extension is a very competitive approach to provide explainable classifications of NOTAMs.

The paper is organized as follows. In Section 3.2, we briefly discuss boolean decision rules generation and provide an overview of binary classification using the boolean decision rules framework of [16]. In Section 3.3, we describe how we extend the binary classification framework to a multiclass classification framework. In Section 3.4, we describe the computational approach for the classification, while in Section 3.5, we present the evaluation of our approach for the NOTAM classification problem. Finally, Section 3.6 draws some conclusions and describes future research objectives.

3.2 Overview of Boolean Decision Rules and Other Classification Methods

Boolean decision rules are easy to interpret. The satisfied clauses can be used to explain a prediction. For multiclass classification, decision trees provide the most interpretable classification. In fact, to the best of our knowledge, there are no other multiclass classification methods that are interpretable or explainable. So, besides comparing the approach that we propose to decision trees, we also benchmark it against some standard, but non-explainable, machine learning algorithms for multiclass classification: Bernoulli Naive Bayes, Neural network, XGBoost [63], Linear Support Vector Machine [54]. Among these algorithms, only Support Vector Machine is a binary classification method and is extended to a multiclass classification method using the one-vs-rest approach. Hence, we survey some of the alternative ways to generate boolean decision rules and describe the approach used in [16] after that. Our extension to multiclass classification can be used for all of them since they all output a boolean decision rule.

Boolean decision rules can be generated in many ways. We selected the approach in [16] because it focuses both on the rule complexity and the accuracy, thus prioritizing interpretability. Some of the alternatives for learning boolean decision rules are as follows. The AQ method [27] attempts to learn the rule by considering one clause at a time heuristically. The clauses only cover the positive examples. The AQ method does not work if the dataset has noise (examples with same features belonging to different classes). The CN2 method [28, 29] extends the AQ algorithm and also generates clauses that may cover a limited number of negative examples. It generates the clause leveraging ideas from the ID3 algorithm [30]. More precisely, ID3 is used for selecting decision tests in a node of a decision tree. The ID3 algorithm measures the impurity of a node in the decision tree by an entropy function. The decision test that minimizes the impurity is selected for each node. Similarly, the CN2 algorithm computes the impurity of the clauses (using the entropy function) to evaluate their quality. The clauses that cover mostly the positive examples and a few negative examples are less impure. Such clauses are added to the decision rules. This allows the CN2 algorithm to work with the data with noise. The RIPPER algorithm [31] learns the rules by using a separate-and-conquer strategy. It starts by treating the least common class in the dataset. It generates the clauses by using a general-to-specific method for growing clauses. Then, it prunes the generated clauses depending on their performance in the validation set. These algorithms mostly aim for accuracy and do not focus on reducing the rule complexity. Hamming clustering [32] is another way of learning boolean rules. Here, the rules are learned by clustering the inputs and then pruning to generalize. A detailed survey for rule learning is given in [33].

In [16], the authors describe a simple MIP model to construct a DNF boolean decision rule. In this model, they use binary variables to represent clauses. The clause is included in the decision rule if the corresponding binary variable takes value 1. We also want our decision rule to be interpretable. A decision rule that contains too many clauses or too large clauses is less interpretable. To avoid generating such rules, they use a hyperparameter C for limiting the complexity of the decision rule. The complete model is based on the notation presented in Table 3.1.

Table 3.1 Model description: sets, parameters, and variables.

Sets	
$\mathcal{P}$	Set of all positive examples in the dataset.
$\mathcal{Z}$	Set of all negative examples in the dataset.
$\mathcal{K}$	Set of all clauses.
$\mathcal{K}_i$	Set of all clauses satisfied by example $i \in \mathcal{P} \cup \mathcal{Z}$.
Parameters	
c_k	Complexity of clause $k \in \mathcal{K}$. Typically, it is set to 1 plus the length of the clause.
C	Complexity of the explanation.
Decision Variables	
w_k	Binary variable indicating if clause $k \in \mathcal{K}$ is selected.
ξ_i	Binary variable indicating if example $i \in \mathcal{P}$ does not satisfy any selected clauses.

The resulting MIP reads as follows:

$$z_{MIP} = \min \quad \sum_{i \in \mathcal{P}} \xi_i + \sum_{i \in \mathcal{Z}} \sum_{k \in \mathcal{K}_i} w_k \tag{3.1a}$$

$$s.t. \quad \xi_i + \sum_{k \in \mathcal{K}_i} w_k \geq 1, \quad \forall i \in \mathcal{P} \tag{3.1b}$$

$$\sum_{k \in \mathcal{K}} c_k w_k \leq C \tag{3.1c}$$

$$\xi_i \geq 0, \quad \forall i \in \mathcal{P} \tag{3.1d}$$

$$w_k \in \{0, 1\}, \quad \forall k \in \mathcal{K}. \tag{3.1e}$$

The objective function (3.1a) minimizes the sum of the number of positive examples that do not satisfy any selected clauses (first term) and the number of positively-classified negative examples (second term). Constraints (3.1b) ensure that each positive example i satisfies at least one of the selected clauses. If not, the corresponding penalty variable ξ_i is set to 1. Indeed, the ξ variables are implied binary, so they do not need to be recorded as binary variables. The constraint (3.1c) limits the overall complexity of the generated decision rule,

where the individual clause complexity is set to c_k. Finally, note that the objective minimizes a total penalty, where the penalty for a positive example is given by its corresponding ξ variable and that for a negative example is set to the total number of selected clauses satisfied by that example. In a column generation context, we refer to model (3.1) as the integer master problem (integer MP) and its linear relaxation as the master problem (MP).

Notice that in the MP, we have a variable for each clause. The number of possible clauses is exponential in the number of features. For this reason, we cannot solve the MP as it is. In [16], the authors use column generation for solving the MP. In this process (see, e.g., [64]), the MP is initialized by the ξ variables and a small number of w variables. We refer to this linear program as a restricted master problem (RMP). Then, the w variables are generated (priced) by solving a subproblem (SP) and added to the RMP as needed, i.e., when they have a negative reduced cost with respect to the dual solution of the current RMP.

More precisely, the reduced cost of column w_k is given by

$$\sum_{i \in \mathcal{Z}} \delta_i - \sum_{i \in \mathcal{P}} \mu_i \delta_i - \lambda c_k,$$

where $\mu_i \geq 0$ is the dual variable of the i^{th} constraint (3.1b), $\lambda \leq 0$ is the dual variable of constraint (3.1c), and the binary variable δ_i is set to 1 if the example $i \in \mathcal{P} \cup \mathcal{Z}$ satisfies the clause w_k. Using the notation provided in Table 3.2, the SP used to generate clause variables w_k can be formulated as follows for a given RMP optimal dual solution (λ^*, μ^*):

Table 3.2 Subproblem description: sets, parameters, and variables.

Sets	
J	Set of all features.
$S_i = \{j \in J : x_{ij} = 0\}$	The zero-valued features in sample $i \in \mathcal{P} \cup \mathcal{Z}$.
Parameters	
D	Maximum size of the generated clause.
Decision Variables	
z_j	Binary variable indicating if feature j is present in the generated clause.
δ_i	Binary variable indicating if example i is satisfied by the generated clause.

$$z_{CG} = \min \quad \sum_{i \in \mathcal{Z}} \delta_i - \lambda^* \left(1 + \sum_{j \in J} z_j\right) - \sum_{i \in \mathcal{P}} \mu_i^* \delta_i \tag{3.2a}$$

$$s.t. \quad \delta_i + z_j \leq 1, \quad \forall j \in S_i, i \in \mathcal{P} \tag{3.2b}$$

$$\delta_i \geq 1 - \sum_{j \in S_i} z_j, \quad \forall i \in \mathcal{Z} \tag{3.2c}$$

$$\sum_{j \in J} z_j \leq D \tag{3.2d}$$

$$z_j \in \{0, 1\}, \quad \forall j \in J \tag{3.2e}$$

$$\delta_i \geq 0, \quad \forall i \in \mathcal{P} \cup \mathcal{Z}. \tag{3.2f}$$

In the objective (3.2a), the term $1 + \sum_{j \in J} z_j$ is equal to the generated clause complexity, i.e., to 1 plus the clause length. This objective function corresponds to the reduced cost of the clause generated. A negative objective value identifies a clause with a negative reduced cost, which can be added to the RMP. The clause is constructed by selecting the features for which the corresponding z variables are set to 1. The constraints (3.2b) – (3.2c) relate the selected features with the δ variables. Constraint (3.2d) controls the size of the clause being generated. Here another hyperparameter D is used. Clearly, the value of D should be less than or equal to the value of C. Note that the variables δ_i are implied binary, so they need not be encoded as binary. Indeed, constraints (3.2b) (resp. (3.2c)) are valid for all examples, but they become relevant only for the positive (resp. negative) ones because of the sign in the objective function.

Column generation stops when no more negative reduced cost columns can be generated or when a maximum number of column generation iterations is reached. In the former case, the cost of the last RMP solution provides a lower bound on the optimal value of the integer MP. In the latter case, we do not have this guarantee. For solving the integer MP, we can use branch-and-price that embeds column generation within a branch-and-bound search tree (see [9]; [7]) . Instead, we just solve the integer version of the final RMP using a commercial MIP solver, yielding a so-called restricted master heuristic [10].

Overall, the process is as follows. We start with a training dataset. The integer MP (3.1) is then solved as described above. The computed solution gives us a DNF decision rule. We use this decision rule to classify new unseen inputs. If an input is classified as positive, the clauses that are satisfied by that example in the DNF rule serve as an explanation for "why the input was classified as positive?".

3.3 Extending the Model for Multiclass Classification

In this section, we extend the technique described in [16] for explainable binary classification to our multiclass classification problem. Multiclass classification can be done by extending the binary classification framework to one-vs-rest classification. In the one-vs-rest classification, we train the binary classification model for each class in the input dataset. For each model, we use the training data inputs belonging to other classes as negative examples and inputs belonging to the class being trained as positive examples.

So, the overall training process is as follows. Consider a Qcode, we mark the inputs as positive or negative depending on whether they have that Qcode or not. Then, we solve the integer MP (3.1) using column generation and the restricted master heuristic. This procedure outputs a DNF rule as our explanation for that Qcode. We repeat this process for each Qcode and store the explanations. This ends our training phase. For evaluation, we consider an unseen input (message). We evaluate this message on each of the stored explanations (DNF rules). Then, we output a list of Qcodes for which the DNF rules evaluated to TRUE on this input. This list is called the candidate list.

3.3.1 Change in the Objective Function

One of the major challenges is about handling the imbalance in the training data for each class. Note that the model above is designed to optimize accuracy. In our task, for each Qcode, there are very few positive examples compared to the number of negative examples. Hence, optimizing for accuracy leads to many false negatives, which is not desirable. Even if we limit the number of negative examples to match the number of positive examples, we cannot directly tune the recall (number of identified positives divided by total number of positives). Furthermore, using very few negative examples leads to a poor formula that does not generalize well for the unseen examples. One way to address this issue is to introduce a new hyperparameter P that controls the recall during training. We change the objective function (3.1a) to have increased penalty for the false negatives by increasing the coefficient of each ξ_i variable to P. The new objective is given by

$$z_{MIP} = \min \quad P \sum_{i \in \mathcal{P}} \xi_i + \sum_{i \in \mathcal{Z}} \sum_{k \in \mathcal{K}_i} w_k. \tag{3.3}$$

3.3.2 Candidate Weights

Another of the challenges with the one-vs-rest classification method is picking a winner when more than one class shows positive results for the membership. This may result in a long candidate list. This is a common challenge across each classifier that extends the binary classification to multiclass classification using one-vs-rest methods. Linear support vector machines (SVM) is a good example for this. Linear SVM is a binary classification method. In the Scikit-learn [65] linear SVM implementation, the winner is picked by using the distance between the data point and the decision boundary (a hypersurface that separates the two classes). For linear SVM, the decision boundary is a line separating the two classes. The distance between a data point and the decision boundary indicates how certain the model is about the prediction. When more than one class shows positive results, the class, for which the distance to the decision boundary is maximized, is picked as winner.

If we train for a small number of Qcodes (say, up to 10), then the mean candidate list length is not large. However, we face the issue of large candidate list lengths when we train for a large number of Qcodes (at least 100). This motivates us to eliminate some of the redundant candidates from the list. Unlike SVM, we do not have a distance function because our decision boundary is a discrete function. However, we can assign some weight to each candidate in the list in a different way such that a higher weight corresponds to a higher likelihood of the example belonging to that class. We then sort the candidates in decreasing order of their weights and keep only the first few candidates with the highest weights. The candidate weights are computed as follows.

For each Qcode, we compute the explanation by solving the integer MP (3.1). Recall that each explanation is an OR of clauses (DNF). Now consider a clause in the explanation. We want to assign a weight to this clause so that it reflects how strongly it indicates whether the example belongs to that candidate or not. We use two numbers to compute the weight for a clause, namely

1. Positive accuracy W_p: Number of positive training examples that satisfy the clause divided by the total number of positive examples in the training data.

2. Negative accuracy W_n: Number of negative training examples that satisfy the clause divided by the total number of negative examples in the training data.

Naturally, we want to assign a high weight to a clause for which W_p is large and W_n is small. We tried a few different ways for computing the weights combining W_p and W_n, and ultimately, we have chosen to set them as $W_{diff} = W_p - W_n$. This combination is simple and

works better than some other combinations like $W_{ratio} = W_p/(W_n + \epsilon)$, where ϵ is a small positive number. These are the reasons we have selected it.

In our data, we do not have the same number of positive examples for each label. A clause that is generated with a small amount of training data gives us less confidence compared to a clause that is generated using a large amount of training data. So, to factor that, we also multiply the weights by the number of total training data used for that Qcode denoted as n_k, thus finally giving $W^*_{diff} = (W_p - W_n) \times n_k$.

Consider a new unseen message and a candidate list. We want to compute the weight for each Qcode in this list. For a Qcode, we know the weights of each clause in the explanation and we only consider the clauses that are satisfied by this new example (there is at least one such clause if the Qcode is in the candidate list). The weight of the Qcode is computed as the sum of the weights of these clauses.

3.4 Computational Approach

This section presents the computational approach for predicting the Qcodes of NOTAM messages. We start by describing the preprocessing of the data in Section 3.4.1. Then, in Section 3.4.2, we discuss the training part, which involves solving a sequence of integer MPs (3.1), namely, one per Qcode. Finally, we describe in Section 3.4.3 how prediction is performed for new unseen NOTAMs.

3.4.1 Data Preprocessing

We first preprocess the data to make it ready for the training task. This requires mainly three tasks: Cleaning, Splitting and Binarization. For this task, our industrial partner Thales gave us a sample dataset for training. The dataset has a total of 21,358 NOTAMs and a total of 603 different Qcodes. It is available at `https://github.com/krooonal/NOTAM_explainable_prediction_data`.

Cleaning: First, we discard all the Qcodes for which we do not have a sufficient number of positive examples to avoid poor generalization. We keep only the Qcodes for which there are at least 10 positive examples in the dataset. After removing the Qcodes with less than 10 positive examples, we are left with 20,223 NOTAMs and 204 Qcodes. This step eliminates a lot of Qcodes, but the same approach can be extended to generate explanations for the eliminated Qcodes, if we get more data. Moreover, for this prediction task, we only use the message field of the NOTAMs as input and Qcode field as output. We discard all the other information.

In the NOTAMs, we remove all the punctuation characters except '/' (since it is very frequent) and replace them by a space. All the messages are already in capital letters. No other preprocessing is done on the input.

Splitting: We split the data into three sets: Training (60%), Validation (20%), and Test (20%). Training data is used for training, Validation data is used for tuning the hyperparameters and Test data is used for estimating the performance of the model.

Binarization: The MP (3.1) requires the data to have binary features. So, we convert the data into a *binary bag of words* representation as follows. We iterate through the Training dataset and select the most frequent 1,000 words. The number of words is a hyperparameter and can be further tuned. However, we observe that using more words does not justify the improvement compared to the increase in training time for our data. Those most frequent words are our binary features for our input: for each word, we have an input binary entry taking value 1 if the input NOTAM contains that word and 0 otherwise. Note that with this representation, we lose the information about the order in which the words appear. This can be an issue for many text classification applications, but in our case the messages are short and simple, so the prediction with high accuracy can still be done with this representation.

3.4.2 Training

To determine the DNF rule for each Qcode, the integer MP (3.1) is solved considering the training dataset. To do so, we use column generation and the restricted master heuristic as described in Section 3.2, with the following specificities.

Column generation stops when no more negative reduced cost column can be found or when a maximum number of column generation iterations I is reached, where one iteration corresponds to solving the RMP and the SP once each. I is a hyperparameter.

Since the training data is large, we do not include all the negative examples while training for a particular Qcode. We randomly select a fraction of negative examples as $N(\geq 1)$ times the number of positive examples. Hence, we also have N as a hyperparameter.

In the SP (3.2), D is a hyperparameter that limits the size of the clause generated. The value of D should be clearly less than the value of C in the integer MP (3.1). When generating clauses with $D = C$, we find that the generated clause size never exceeds 3. So, we use $D = 3$. Choosing a small value for D helps to reduce the training time.

In terms of solving time, as shown in Table 3.4, the real bottleneck is associated with solving SP (3.2). To reduce the training time, we use a heuristic to find negative reduced cost clauses. For each Qcode, we start by selecting at most 200 of the most frequent words in the positive

examples. Then, we remove all the words that appear in less than 2% of the examples. Using this small subset of words, we compute all size 1 and size 2 clauses and their reduced costs. We add all the clauses with negative reduced costs to the RMP. We only solve the SP (3.2) if we fail to find any clause with negative reduced cost with this heuristic.

For solving SP (3.2), we observed that the OR-tools (version 9.2) solver CP-SAT [66], a constraint programming solver using a SAT solver as back-end for tree search, is approximately 20% faster than Gurobi [67] (version 9.1.1). This is probably because of the combinatorial structure of the SP. For this reason, the solving approach in CP-SAT that uses significantly less LP solving is beneficial. However, CP-SAT can only accept integer coefficients. So, to use it, we have to round the dual values to integer values: we round all the coefficients in the objective (3.2a) up (i.e., all the dual variables λ^* and μ^* are rounded down). So, the columns generated with CP-SAT (the ones with negative reduced cost) are always valid. To minimize the error due to rounding, we scale the objective coefficients by multiplying them by 100. In addition, to avoid overfitting, we only add the columns that have reduced cost less than -10^{-2}. To achieve a faster convergence, we add all the negative reduced cost columns we find during the solving process, as this number of columns is currently not critical in our application.

Notice that while using CP-SAT, we explicitly require all implicitly-binary δ variables to be binary. Since solving SP is highly time-consuming, we use 8 threads with the CP-SAT solver, which is configured with the default parameter values.

To further reduce the search space in SP (3.2), we fix to 0 all the z variables that correspond to words that do not appear in any of the positive examples. This removes many feasible solutions of SP (3.2), but preserves all the solutions that correspond to negative reduced cost clauses. This is because only the last term in the SP objective (3.2a) has negative coefficients. So, at least one positive example has to satisfy the generated clause in order to have a negative reduced cost.

For solving the RMP at each column generation iteration, we use GLOP, an open-sourced linear programming solver in Google OR-tools, version 9.2 [66], with a single thread. For solving the final RMP with integrality requirements, we use Gurobi, version 9.1.1 [67].

For all the training process, we used a CPU server with 2 sockets of Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz 28 cores each (total of 56 cores) and 512G of RAM. However, for training, we used at most 8 cores.

3.4.3 Output

At the end of training, we have an explanation in DNF form for each Qcode. We use these DNF rules for classifying the new data. We check the message against each explanation and output the list of Qcodes for which the message evaluates to TRUE in the corresponding DNF formula. This is our candidate list for that message. As described in Section 3.3.2, we prioritize the Qcodes in the candidate list using candidate weights. We then select the top K ($1 \leq K \leq 10$) candidates (Qcodes) from the list and discard the other candidates. The resulting candidate list, along with the candidate weights, is passed to a human (domain expert) to select the final QCode.

To evaluate the performance of our approach, we measure the accuracy and the average list length for each reduced candidate list and also for each complete candidate list.

3.5 Computational Results

This section presents the computational results obtained on the real data provided by our industrial partner Thales (see Section 3.4.1).

3.5.1 Tuning Experiments

First, Table 3.3 reports experimental results obtained from complete target lists when varying the values of the hyperparameters listed in the first four columns. For each hyperparameter configuration, we record the total training time (in seconds), accuracy, maximum candidate list length and average candidate list length per Qcode. For each example, if the candidate list contains the actual Qcode, we count it as a successful prediction. The accuracy is measured by the number of successful predictions divided by the total number of examples evaluated. As seen in the Table 3.3 first six rows, it is easy to achieve high accuracy with very high candidate list length. The accuracy drops as the parameters make the algorithm more selective to reduce the candidate list length. Also note that, while the average candidate list length in the last three rows of Table 3.3 is small enough to be interpreted by humans, the maximum candidate list length is very high. To overcome this, we need to reduce the list length by removing the candidates with lower weights W_{diff}.

Table 3.3 Hyperparameter tuning experimental results

P	C	I	N	Training time (seconds)	Training			Validation		
					Accuracy	List length		Accuracy	List length	
						Max	Avg		Max	Avg
2	10	10	2	916	0.96	72	9.93	0.92	72	9.97
2	10	20	2	988	0.96	73	10.21	0.92	70	10.28
2	10	30	2	993	0.96	71	10.03	0.92	71	10.02
3	10	30	2	1025	0.97	78	11.35	0.93	75	11.35
4	10	30	2	1035	0.98	78	12.22	0.94	78	12.29
5	10	30	2	1028	0.98	80	12.48	0.94	77	12.56
4	10	30	5	1297	0.95	60	7.53	0.91	60	7.57
4	10	30	10	2006	0.93	50	5.15	0.89	49	5.14
4	10	30	20	3356	0.91	46	3.47	0.86	46	3.48
4	20	30	20	4488	0.94	46	3.15	0.87	46	3.14
4	30	30	20	5381	0.95	47	3.05	0.87	46	3.02

Table 3.4 shows various statistics related to training for various parameter configurations (identified by the first four columns). We can see that most of the time is used for solving the SP using the CP-SAT solver. Compared to that, the time required to solve the RMP and the integer RMP is very small. We see from the last three rows that the average clause length per Qcode increases with C as expected. In the fifth column of Table 3.4 ('Number of MPs solved'), we record the number of times (out of 204 Qcodes) for which the MP is solved, i.e., where the subproblem cannot find any column with negative reduced cost within a tolerance of 10^{-2}. We also record the number of columns added by the heuristic (described in Section 3.4.2) and by CP-SAT. This helps us measure the effectiveness of the heuristic. Table 3.4 also shows the average clause length per Qcode and the time taken by various parts of our column generation approach.

From the results in Tables 3.3 and 3.4, the effect of increasing each hyperparameter value can be summarized as follows.

P: Improves accuracy and increases list lengths.

C: Improves accuracy, reduces list lengths, and increases average clause length.

I: For more Qcodes, the MPs are solved before reaching the maximum number of itera-

Table 3.4 Hyperparameter tuning training statistics

				Number of MPs solved	Average number of columns added per Qcode		Average clause length per Qcode	Average Time per Qcode (in seconds)				
P	C	I	N		Heuristic	CP-SAT		RMP	CP-SAT	Heuristic	Integer RMP	Total
2	10	10	2	181	14.90	1.58	1.37	0.003	3.44	1.00	0.01	4.49
2	10	20	2	203	15.11	1.64	1.37	0.003	3.63	1.17	0.01	4.84
2	10	30	2	204	14.80	1.74	1.35	0.003	3.74	1.08	0.01	4.87
3	10	30	2	204	16.06	2.08	1.37	0.003	3.88	1.09	0.01	5.03
4	10	30	2	204	15.84	1.86	1.37	0.003	3.93	1.10	0.01	5.07
5	10	30	2	204	15.71	2.23	1.36	0.003	3.87	1.12	0.01	5.04
4	10	30	5	204	13.71	3.56	1.49	0.003	5.51	0.78	0.01	6.36
4	10	30	10	204	10.36	5.74	1.55	0.004	9.18	0.55	0.02	9.84
4	10	30	20	204	5.74	8.97	1.66	0.004	16.04	0.24	0.02	16.45
4	20	30	20	204	5.96	12.23	1.88	0.005	21.48	0.32	0.02	22.00
4	30	30	20	203	5.80	13.80	1.95	0.006	25.81	0.36	0.02	26.38

tions. We observe that for our tests a maximum of $I = 30$ column generation iterations is sufficient to solve the MP for at least 203 of the 204 Qcodes. However, there is no clear signal on accuracy and list lengths. Increases solving time (see the first three rows of Table 3.3)

N: Reduces accuracy, reduces list length, increases average clause length, and increases solving time.

Second, Table 3.5 presents the results for reduced target lists, by considering different maximum length values $K \in \{1, \ldots, 10\}$. For this experiment, we fix what turns out to be the best configuration of parameters, namely, $P = 4$, $C = 30$, $I = 30$, and $N = 20$.

Table 3.5 Accuracy for the hyperparameter configuration ($P = 4, C = 30, I = 30, N = 20$) by using different candidate list lengths K. The results are compared with the last line of Table 3.3 denoted as $K = \infty$.

K	Training	Validation
1	0.65	0.60
2	0.86	0.80
3	0.90	0.84
4	0.92	0.85
5	0.93	0.86
6	0.94	0.86
7	0.94	0.87
8	0.94	0.87
9	0.95	0.87
10	0.95	0.87
∞	0.95	0.87

From the results in Table 3.5, we can observe that we do not achieve good accuracy for very small values of K, but the candidate lists with $K \geq 3$ produces good accuracy values, while $K = 9$ already matches the accuracy in Table 3.3, where K is not limited. It will be shown in Table 3.9 that already $K = 3$ gives results that are comparable to other machine learning algorithms.

Finally, for the best configuration of parameters, i.e., $P = 4$, $C = 30$, $I = 30$, and $N = 20$, we summarize the results in Table 3.6, including the test accuracy with and without limited

target lists. The first column shows the number of examples that had more than one Qcodes in the candidate list. Furthermore, the second part of the table considers the case in which we restrict the evaluation to the Qcodes with at least 50 examples in the training data. As expected, we observe higher accuracy and lower candidate list lengths with the restricted evaluation. This shows that using more data for each Qcode can improve the accuracy and reduce the candidate list lengths.

Table 3.6 Results for best hyperparameter values. The last three lines consider only the Qcodes with at least 50 examples.

Dataset	Number of NOTAMs with more than one Qcode in the list	Average list length	Accuracy	Accuracy ($K = 4$)
Training	8335	3.05	0.95	0.92
Validation	2749	3.02	0.87	0.85
Test	2760	3.01	0.87	0.86
Training	3758	1.49	0.94	0.94
Validation	1235	1.46	0.90	0.89
Test	1206	1.44	0.90	0.89

As an example, we present the boolean decision rule explanation generated for the Qcode 'LAAS':

. ['U/S' AND 'ALS'] OR

. ['RWY' AND 'U/S' AND 'MAINT'] OR

. ['U/S' AND 'SYSTEM' AND 'APCH'] OR

. ['U/S' AND '24'] OR

. ['U/S' AND 'MAINT' AND '02'] OR

. ['RWY' AND 'U/S' AND '19'].

The first part of the Qcode 'LA' indicates 'Approach lighting system (specify runway and type)' and the second part of the Qcode 'AS' means 'Unserviceable'. The positive and negative

accuracies for the clauses are shown in Table 3.7. For this Qcode, we used 3003 data points for training. We can see that the last four clauses are only selected to cover a few positive training data points. This is where the model tries to overfit the training data. We see that the weights for these clauses are very low compared to the other two clauses.

Table 3.7 Weights for clauses for the boolean decision rule for label 'LAAS'

Clause	Positive accuracy	Negative accuracy	W^*_{diff}
'U/S' AND 'ALS'	0.937	0.006	2795.79
'RWY' AND 'U/S' AND 'MAINT'	0.035	0.002	99.10
'U/S' AND 'SYSTEM' AND 'APCH'	0.007	0.000	21.02
'U/S' AND '24'	0.007	0.000	21.02
'U/S' AND 'MAINT' AND '02'	0.007	0.000	21.02
'RWY' AND 'U/S' AND '19'	0.007	0.000	21.02

3.5.2 Comparison with Standard ML Models

In this section, we compare the accuracy of our approach with those that can be achieved by using the following standard machine learning algorithms.

1. Max Class Classifier: Picks the most frequent class in the training data as the prediction. This is typically used for benchmarking.

2. Bernoulli Naive Bayes: This classifier assumes that the data comes from a Bernoulli distribution. It uses the training data to find the parameters for the underlying Bernoulli distribution. Finally, it uses Bayes' rule to estimate the probability of an example belonging to a fixed class.

3. Decision trees: This classifier builds a tree. Each internal node of this tree has a split check. The data point follows the branch corresponding to the split check decision. Finally, each leaf node is assigned a class. The data point is classified into the class of the leaf node it reaches. This is the most interpretable classifier among the ones compared.

4. XGBoost: This is a combination of multiple simple classifiers (like decision trees) that are trained sequentially.

5. Linear SVM: This is a binary classifier that separates the datapoints of the two classes by a line in such a way that the distance of the nearest datapoint from the line is

maximized. This is the only classifier that is designed for binary classification and extended to multiclass classification.

6. ReLU neural network: ReLU is a piecewise linear function defined as $ReLU(x) = max(0, x)$. The neural network passes the input through a series of non linear functions ($ReLU(wx + b)$) in order to predict the final class. The goal is to be able to learn a much more complex function.

The hyperparameters for all the models except XGBoost were tuned using Scikit-learn's GridsearchCV method. For XGBoost, we used Scikit-learn's RandomizedSearchCV method because of the higher training time. The values that achieved the highest accuracy on the validation set were selected. We report the values considered for each hyperparameter in Table 3.8.

Table 3.8 Hyperparameter tuning for the standard ML algorithms.

Model	Hyperparameter (Scikit-learn)	Values considered	Selected value
Bernoulli Naive Bayes	α	0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1	0.04
Decision trees	criterion	gini, entropy	gini
	max depth	4, 8, 16, 32, 64, 128	64
	max features	sqrt, log2	sqrt
	max leaf nodes	4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048	2048
	min samples leaf	1, 11, 21, 31, 41, 51	1
	min samples split	2, 12, 22, 32, 42	12
	splitter	best, random	random
XGBoost	colsample bytree	All reals in (0.7, 1.0)	0.75
	gamma	All reals in (0,0.5)	0.179
	learning rate	All reals in (0.03, 0.33)	0.255
	max depth	All integers in [2 ,6]	5
	n estimators	All integers in [100, 150]	112
	subsample	All reals in (0.6, 1.0)	0.828
	tree method	hist	hist
	objective	multi:softprob	multi:softprob
Linear SVM	C	0.01, 0.1, 1.0, 10.0, 100.0	1.0
	max iter	10000000	10000000

We used the python Scikit-learn library for evaluating the results that were generated on Google Colab (free version) and are reported in Table 3.9, where K denotes the candidate list length. The last three rows in this table provide the results of the proposed Boolean decision rule approach, using the best hyperparameter configuration ($P = 4, C = 30, I = 30, N = 20$) for three maximum list lengths $K = 3, 4, 5$, respectively.

The results in Table 3.9 show that XGBoost gives the best accuracy in the test data followed by a (relatively) simple 5-layer ReLU neural network and a Linear SVM for single target prediction ($K = 1$). For these models, the best accuracy on the test data is comparable to the accuracy achieved on validation and test data by our column generation approach. However, decision trees, which are more interpretable than SVM and neural networks, cannot match the accuracy achieved by our proposed prediction, thus corroborating it. For a fair comparison, we also report the accuracies for XGBoost, Linear SVM, and 5-layer ReLU neural network with top $K(> 1)$ candidate predictions instead of a single QCode prediction. These models output a prediction probability for each class. We can extract the top K predictions using these probabilities. In this setting, the column generation approach is still comparable to Linear SVM in terms of accuracies. However, XGBoost and neural network are significantly better than the column generation approach, but they lack interpretability, the main goal of our investigation.

We did not consider larger neural networks for the comparision. However, Thales reported that they could get 93.9% accuracy (without explanations) using BERT [68], a more advanced machine learning algorithm for text classification and a lot more data (about 100,000 examples).

3.5.3 Practical Impact

The column generation algorithm is not able to match the BERT accuracy result, but it provides an explanation for every NOTAM classification. One of the major issues is, however, the scalability of the algorithm. We note that some of the algorithms for Boolean decision rule generation that are based on heuristics (for example, RIPPER from [31]) require less training time. An interesting venue for future development is to use such methods to generate the starting columns for the MP (3.1).

From an operational perspective, it is crucial for Thales to capture explanations for NOTAM classification, as explanations do contain information essential to personnel concerned with flight operations. Prior to the current work, this was not accessible in any way, while communicating explanations of NOTAM classification is now possible, and useful for validating NOTAM categories. The loss in accuracy has been strongly balanced by the explanation,

Table 3.9 Results of different ML models for multiclass classification.

Model	K	Accuracy		
		Training	Validation	Test
Max Class Classifier	1	0.06	0.06	0.06
Bernoulli Naive Bayes	1	0.84	0.77	0.78
Decision trees	1	0.84	0.76	0.75
XGBoost	1	0.94	0.85	0.85
	3	0.97	0.89	0.89
	4	0.97	0.89	0.89
	5	0.97	0.89	0.89
Linear SVM	1	0.94	0.84	0.84
	3	0.95	0.86	0.86
	4	0.95	0.86	0.86
	5	0.95	0.86	0.86
ReLU Neural Network	1	0.95	0.84	0.84
	3	0.98	0.88	0.88
	4	0.98	0.88	0.88
	5	0.99	0.88	0.88
Boolean Decision Rule	3	0.90	0.84	0.84
	4	0.92	0.85	0.86
	5	0.93	0.86	0.86

which in turn is used for validation, providing very useful insight for human to trust the classification results, and strongly support flight operation.

3.6 Conclusion and Future Work

In this paper, we extended the work of [16] to a real-world multiclass classification problem. We could generate interpretable boolean decision rules for classifying NOTAMs into Qcodes. In the process, we demonstrated a method to tackle the issue of multiple candidate prediction that results from one-vs-rest extension of the binary classification algorithm in [16]. The accuracy we achieved using this method were comparable to some of the state-of-the-art machine learning algorithms like SVM and small neural networks.

We showed a method to generate candidate weights that help us find more relevant candidates when more than one candidate is predicted. Although the way we generate the candidate weights seems reasonable, it does not necessarily give the best weights, as seen in Table 3.5. We plan to work on finding improved ways to compute the candidate weights to get even

better accuracy for smaller candidate lists, keeping in mind that the method for computing the weights should be easy to interpret by humans.

Finally, we also want to explore the effect of using the complete branch-and-price algorithm to generate the explanation instead of only generating columns in the root node. This will obviously result in longer training times, but can potentially help us find better Boolean decision rules. Such rules might then be studied to understand the underlying limitations of classifying NOTAMs using Boolean decision rules.

CHAPTER 4 ARTICLE 2: AN IMPROVED COLUMN-GENERATION-BASED MATHEURISTIC FOR LEARNING CLASSIFICATION TREES

Authors: Krunal Kishor Patel, Guy Desaulniers, and Andrea Lodi

Submitted on August 21, 2023 and published on February 10, 2024 in Computers & Operations Research, Volume 165, DOI: 10.1016/j.cor.2024.106579.

Abstract

Decision trees are highly interpretable models for solving classification problems in machine learning (ML). The standard ML algorithms for training decision trees are fast but generate suboptimal trees in terms of accuracy. Other discrete optimization models in the literature address the optimality problem but only work well on relatively small datasets. [1] proposed a column-generation-based heuristic approach for learning decision trees. This approach improves scalability and can work with large datasets. In this paper, we describe improvements to this column generation approach. First, we modify the subproblem model to significantly reduce the number of subproblems in multiclass classification instances. Next, we show that the data-dependent constraints in the master problem are implied, and use them as cutting planes. Furthermore, we describe a separation model to generate data points for which the linear programming relaxation solution violates their corresponding constraints. We conclude by presenting computational results that show that these modifications result in better scalability.

Keywords: Machine Learning, Decision trees, Column Generation, Classification, Mixed Integer Programming.

4.1 Introduction

In machine learning (ML), a classification problem consists in predicting, from a predefined set of classes, the class to which a data point belongs. Each data point (also called data row) is described by features that are used to predict its class. Classification problems can be solved using decision trees that are highly interpretable models. In a decision tree, each internal node contains a test based on the dataset's features. In this work, we focus on univariate binary decision trees. In such trees, the internal node tests, called hereafter split checks, use only a single feature that can vary from one node to another. Each internal node

has two branches. Each leaf node is associated with a target class. A data row starts at the tree's root node and follows the branches based on its feature values and the split checks at the internal nodes. Finally, it reaches a leaf node, where it is classified in the target class associated with that leaf node.

To determine the split checks of a decision tree, a supervised learning algorithm is run on a training dataset, where the real class of each data point is known. Usually, the goal of this algorithm is to maximize accuracy, i.e., the number of training data points correctly classified. After the training is completed, the decision tree with its selected split checks can be used to classify unseen data rows. The standard ML algorithms for training decision trees, like CART [37] and ID3 [30], use heuristics that optimize for one-depth level accuracies. They are fast in terms of training times. However, the generated trees are suboptimal in terms of overall accuracy.

Learning optimal binary decision trees is an NP-complete problem [40]. Many authors have created optimization models for learning decision trees optimal in terms of accuracy. [41], [42], [43], [44], and [45] introduced Mixed Integer Linear Programs (MIPs) for training decision trees. For a recent survey on continuous and MIP models for training decision trees, refer to [46]. [47] described a mixed integer quadratic model for learning optimal trees. [48,49] introduced mixed integer non-linear models for training decision trees. [50] proposed a continuous optimization approach for training randomized decision trees. [51] introduced a dynamic-programming-based exact algorithm to compute optimal decision trees. [52] presented a SAT model to train decision trees. Finally, [53] proposed a constraint programming model for training decision trees for binary classification tasks.

These models can learn trees with better accuracies compared to CART and ID3. Sometimes they can find the optimal decision tree for a given dataset. However, the biggest issue with these models is scalability. [1] stated that [44] and [42] MIP models failed to handle datasets with more than 10,000 rows. Except for the binary classification model of [53], no other paper (from the ones cited above) showed results on datasets with more than 10,000 rows.

To address the scalability problem, [1] introduced a column-generation-based heuristic to train more accurate univariate binary decision trees of predefined depth. In this approach, the master problem uses one variable for each path in the tree, where a path is defined by a set of nodes from the root node to a leaf node, associated with split checks, and a target class at the leaf node. This results in a large number of variables that are generated using subproblems. [1] proposed a subproblem model that requires solving one subproblem for each leaf and each possible target class. Integer solutions are obtained by transforming the restricted master problem of the last column generation iteration into a MIP and solving it

using a MIP solver.

It should be noted that this approach would still be heuristic even if a branch-and-price algorithm was used because not all possible split checks are considered at each node. However, in our experiments, we analyze the optimality of the computed solution for a fixed set of candidate split checks to gain information about the best accuracy that can be achieved using branch-and-price.

In this paper, we improve the [1] column generation approach to make it faster and more scalable. This includes a preprocessing and initialization routine that reduces the size of the master problem and subproblems in the column generation process and results in faster training. Our main contributions are as follows:

1. We introduce a modified subproblem model for the column generation approach with more variables and constraints than the one proposed in [1] but requires fewer subproblems overall (one subproblem per leaf instead of one subproblem per leaf and target class). We show that the updated subproblem model results in faster training.

2. We prove that the data-dependent constraints in the master problem (enforcing that each row in a given dataset reaches a single leaf in the tree) are implied and can, thus, be added to the model as cutting planes only if they are violated.

3. We extend the set of these constraints to unknown rows by providing a separation model that can generate any unlabeled row for which the corresponding constraint is violated.

4. We computationally show the significant benefits of our approach with respect to [1] on datasets from the literature.

This paper is organized as follows. Section 4.2 presents an overview of the column-generation-based heuristic for training decision trees proposed in [1]. In Section 4.3, we describe the proposed modifications to this matheuristic. In Section 4.4, we report the results of various computational experiments to show the effect of our modifications. Finally, in Section 4.5, we present our conclusions and some future research directions.

4.2 Overview of [1] Column Generation Approach

In this section, we describe the column-generation-based heuristic proposed in [1] for generating decision trees. The authors focus on creating univariate binary decision trees of a fixed

depth. The trees can handle multiclass classification, unlike in [41] and [53]. As this is a heuristic, the authors do not claim to generate optimal classification trees. Instead, they try to address the scalability issues in the other optimization models for learning decision trees.

The problem that [1] attempt to solve is as follows. Consider a training dataset with rows in set R, features in set F, and targets in set T. Without loss of generality, assume that each feature is numerical (if not, we can convert it into a numerical feature by using natural numbers or one-hot encoding). The goal is to learn a complete binary classification tree of a given depth that maximizes accuracy on the training dataset. In other words, we want to find the best split checks for each internal node and the best target values for each leaf node in the decision tree. We use N_{int} and N_{lf} to refer to the set of all internal nodes and the set of all leaf nodes, respectively.

Each split check a has two components. A feature $f_a \in F$ and a threshold value $\mu_a \in \mathbb{R}$ for that feature. A row $r \in R$ with value $v_r^{f_a}$ for feature f_a takes the left branch from the node with this split check if $v_r^{f_a} \leq \mu_a$ and takes the right branch otherwise.

One can consider optimizing over all possible split checks at each node. However, this can be computationally very expensive. Instead, [1] used a subset S_j of all possible split checks (called candidate split checks) for each node $j \in N_{int}$. Because of this, the learned tree may not be optimal. Nevertheless, such a tree provides better accuracy than the trees generated, for example, by CART.

To generate the candidate split checks, [1] use the 'Threshold sampling' process. They used 300 runs of the CART algorithm on randomly selected 90% training data rows. For each run, they collect the generated split check for each node. From the collected split checks, they select the most frequent q split checks for each node j and add them to S_j, where $q = \left\lfloor \frac{150}{|N_{int}|} \right\rfloor$ for the root node and $q = \left\lfloor \frac{100}{|N_{int}|} \right\rfloor$ for the other nodes. Finally, they run CART on the entire training dataset. They select all the split checks generated in this run, i.e., they add them to their respective set S_j. Considering these split checks ensures that the final output will have accuracy at least as high as the CART output if the associated model is solved to optimality.

The goal of the learning problem is to find a split check for each node $j \in N_{int}$ from the given set S_j and a target from T for each leaf node to achieve maximum accuracy on the training dataset.

[1] proposed to model this problem as the following MIP, called the integer master problem (integer MP) in the context of a column generation algorithm. Consider the paths from the root node to the leaf nodes in the tree. Each path has a specific split check assigned to each internal node it contains and a specific target assigned to its leaf node. For each path, we

define a binary variable that takes value 1 if the path is selected and 0 otherwise. The MIP ensures that the selected paths are consistent with each other, i.e., they form a tree. Relying on the notation presented in Table 4.1, the MIP is as follows:

$$Max \quad \sum_{l \in N_{lf}} \sum_{p \in DP_l} CP(p) x_p \tag{4.1a}$$

$$s.t. \quad \sum_{p \in DP_l} x_p = 1, \quad \forall l \in N_{lf} \tag{4.1b}$$

$$\sum_{l \in N_{lf}} \sum_{\substack{p \in DP_l: \\ r \in R^l(p)}} x_p = 1, \quad \forall r \in R \tag{4.1c}$$

$$\sum_{\substack{p \in DP_l: \\ s_p(j) = a}} x_p = \rho_{j,a}, \quad \forall l \in N_{lf}, j \in p_{BT}(l) \cap N_{int}, a \in S_j \tag{4.1d}$$

$$x_p \in \{0,1\}, \quad \forall p \in DP_l, l \in N_{lf} \tag{4.1e}$$

$$\rho_{j,a} \in \{0,1\}, \quad \forall j \in N_{int}, a \in S_j. \tag{4.1f}$$

The objective function (4.1a) maximizes accuracy (minimizes the misclassification error). Constraints (4.1b) ensure that exactly one path is selected for each leaf. Constraints (4.1c) force each row to follow exactly one selected path. The known class of the row may not match the target of the followed path. Constraints (4.1d) impose that the selected paths are consistent with respect to split checks on common nodes: if two paths are selected with a common node, the same split check must be assigned to the common node in both paths. Note that in this model, the variable upper bounds are implied and can be dropped while solving the linear relaxation of the MIP.

In general, each row in the training dataset has equal weights. However, this model is able to support rows with different weights. In this case, we need to compute the objective coefficients $CP(p)$ as a weighted sum of the correctly classified rows for each path p.

Model (4.1) contains a large number of variables, one per path in sets DP_l, $l \in N_{lf}$. As proposed by [1], we can alleviate this drawback by applying column generation. In this context, we refer to the linear relaxation of the MIP as the master problem (MP) and to the MIP itself as the integer MP. Column generation is used to solve the MP. At each iteration, it solves using a standard linear programming solver a restricted MP (RMP), i.e., the MP restricted to a small subset of its variables. In the first iteration, the RMP is initialized with the paths generated from the last run of CART on the complete training dataset in the threshold sampling process. Solving the current RMP provides a pair of optimal primal and

Table 4.1 Notation for the MIP (4.1).

Sets	
R	set of rows in the dataset.
F	set of features in the dataset.
N_{lf}, N_{int}	sets of leaf and internal nodes in the decision tree.
$p_{BT}(l)$	set of nodes in the paths to leaf l in binary tree.
DP_l	set of decision paths ending in leaf l.
$R^l(p)$	subset of rows directed to leaf l through path p.
S_j	set of candidate split checks for node j.
Parameters	
$s_p(j)$	split check assigned at node j in path p.
$CP(p)$	number of correct predictions for path p.
Decision Variables	
x_p	binary variable indicating if path $p \in DP_l$ is assigned to leaf $l \in N_{lf}$.
$\rho_{j,a}$	binary variable indicating if split check $a \in S_j$ is assigned to node $j \in N_{int}$.

dual solutions. To verify if the primal solution is also optimal for the whole MP, a set of subproblems (SPs), namely, one for each leaf node $l \in N_{lf}$ and each target class $t \in T$, is solved with the goal of identifying columns (paths) with a positive reduced cost with respect to the current RMP dual solution. When such columns are found, they are added to the RMP and another iteration is started. Otherwise, the column generation process stops with an optimal solution to the MP.

Relying on the additional notation presented in Table 4.2, the SP for leaf $l \in N_{lf}$ and target $t \in T$ can be formulated as the following MIP:

$$Max \quad \sum_{r \in R_t} y_r - \alpha_l - \sum_{j \in p_{BT}(l)} \sum_{a \in S_j} \gamma_{l,j,a} u_{j,a} - \sum_{r \in R} \beta_r y_r \tag{4.2a}$$

$$s.t. \quad \sum_{a \in S_j} u_{j,a} = 1, \quad \forall j \in p_{BT}(l) \tag{4.2b}$$

$$y_r \leq \sum_{a \in S_j \cap T(r)} u_{j,a}, \quad \forall j \in LC(l), r \in R \tag{4.2c}$$

$$y_r \leq \sum_{a \in S_j \cap F(r)} u_{j,a}, \quad \forall j \in RC(l), r \in R \tag{4.2d}$$

$$\sum_{j \in LC(l)} \sum_{a \in S_j \cap T(r)} u_{j,a} + \sum_{j \in RC(l)} \sum_{a \in S_j \cap F(r)} u_{j,a} - (k-1) \leq y_r, \quad \forall r \in R \tag{4.2e}$$

$$\sum_{j \in p_{BT}(l)} u_{j,a} \leq 1, \quad \forall a \in \bigcup_{j \in p_{BT}(l)} S_j \tag{4.2f}$$

Table 4.2 Additional notation for the SP (4.2) defined for leaf l and target t.

Sets	
R_t	set of rows in the dataset with target t.
$LC(l)$	set of nodes in N_{int} that have left child in $p_{BT}(l)$.
$RC(l)$	set of nodes in N_{int} that have right child in $p_{BT}(l)$.
$T(r)$	set of split checks for which row r takes the left branch: $\{a = (f_a, \mu_a) \in \cup_{j \in N_{int}} S_j : v_r^{f_a} \leq \mu_a\}$.
$F(r)$	set of split checks for which row r takes the right branch: $\{a = (f_a, \mu_a) \in \cup_{j \in N_{int}} S_j : v_r^{f_a} > \mu_a\}$.
Parameters	
k	depth of the decision tree, levels are indexed by $h = 0, ..., k-1$.
v_r^f	value of feature f in row r.
α_l	dual value of constraint (4.1b) for leaf l.
β_r	dual value of constraint (4.1c) for row r.
$\gamma_{l,j,a}$	dual value of constraint (4.1d) for leaf l, node j, and split check a.
Decision Variables	
y_r	binary variable indicating if row $r \in R$ reaches leaf l.
$u_{j,a}$	binary variable indicating if split check $a \in S_j$ is assigned to node $j \in p_{BT}(l)$.

$$y_r \in \{0,1\}, \quad \forall r \in R \tag{4.2g}$$

$$u_{j,a} \in \{0,1\}, \quad \forall j \in RC(l) \cup LC(l), a \in S_j. \tag{4.2h}$$

The objective function (4.2a) aims at maximizing the reduced cost of the path generated. Constraints (4.2b) ensure that exactly one split is selected for each node in the path. Constraints (4.2c)-(4.2e) evaluate if the row r would reach the leaf by following the generated path. That is, the variable y_r takes the value 1 if and only if a split check from the set $T(r)$ is selected for all nodes in $LC(l)$ and a split check from the set $F(r)$ is selected for all the nodes in $RC(l)$. Because of the objective direction, constraint (4.2e) for a row r is only needed if $\beta_r \geq 1$ for $r \in R_t$ and $\beta_r \geq 0$ for $r \notin R_t$. Constraints (4.2f) impose that each split is selected at most once in a path. Given that these constraints led to infeasible SPs when the candidate sets of split checks are small for some nodes in our experiments, we have decided to remove them. This only increases the search space but does not change the rest of the algorithm.

Because model (4.2) is a MIP, generating paths using it can be computationally expensive. To address that, [1] also introduced a pricing heuristic. This heuristic randomly generates paths and evaluates their reduced costs. If the reduced cost of a generated path is positive, the path is added to the RMP. Model (4.2) is only used if the heuristic fails to find any path with a positive reduced cost.

In theory, the column generation process stops if model (4.2) proves that all columns have a non-positive reduced cost. Given that the convergence might be slow, it can also terminate when a time limit is reached.

To obtain an optimal integer solution to the integer MP (4.1), non-truncated column generation can be embedded into a branch-and-bound algorithm to yield a branch-and-price algorithm [9]. However, to limit the computational times, [1] rather applied a RMP heuristic [10, 69] that consists in converting the last RMP solved into a MIP that can be solved using a commercial MIP solver without adding any new columns. The selected paths in the final solution describe a valid learned tree.

4.3 Modifications of the Column Generation Approach

This section describes our modifications of the column-generation-based heuristic of [1]. In Section 4.3.1, we show how to reduce the number of SPs by adding the target computation for the leaf nodes in the constraints of the SPs. In Section 4.3.2, we analyze the constraints (4.1c) in the integer MP (4.1) and show that they are implied by the other constraints. In Section 4.3.3, we describe a separation algorithm for using constraints (4.1c) as cutting planes to speed up the solving process.

4.3.1 Merged SPs

In [1], there is one SP (4.2) for each leaf $l \in N_{lf}$ and each target $t \in T$. Consequently, there are $|N_{lf}| \times |T|$ SPs in total. We propose to merge the SPs by incorporating the target computation in the MIP model using extra variables and constraints. This reduces the number of SPs to $|N_{lf}|$. This modification is inspired by the flow-based MIP formulation of [45], which is very similar to (4.2), except that it focuses on the entire tree instead of just one path. Relying on the additional notation presented in Table 4.3, the merged SP for a given leaf $l \in N_{lf}$ is formulated as follows:

$$Max \quad \sum_{r \in R} W_r z_r - \alpha_l - \sum_{j \in p_{BT}(l)} \sum_{a \in S_j} \gamma_{l,j,a} u_{j,a} - \sum_{r \in R} \beta_r y_r \tag{4.3a}$$

$$s.t. \quad \sum_{a \in S_j} u_{j,a} = 1, \quad \forall j \in p_{BT}(l) \tag{4.3b}$$

$$y_r \leq \sum_{a \in S_j \cap T(r)} u_{j,a}, \quad \forall j \in LC(l), r \in R \tag{4.3c}$$

Table 4.3 Additional notation for the SP (4.3) for leaf l.

Parameters	
W_r	weight of row r.
Decision Variables	
z_r	binary variable indicating if row $r \in R$ reaches leaf l and has the same target as the path being generated.
w_t	binary variable indicating if target $t \in T$ is selected for the generated path.

$$y_r \leq \sum_{a \in S_j \cap F(r)} u_{j,a}, \quad \forall j \in RC(l), r \in R \tag{4.3d}$$

$$\sum_{j \in LC(l)} \sum_{a \in S_j \cap T(r)} u_{j,a} + \sum_{j \in RC(l)} \sum_{a \in S_j \cap F(r)} u_{j,a} - (k-1) \leq y_r, \quad \forall r \in R \tag{4.3e}$$

$$z_r \leq y_r, \quad \forall r \in R \tag{4.3f}$$

$$z_r \leq w_t, \quad \forall t \in T, r \in R_t \tag{4.3g}$$

$$\sum_{t \in T} w_t = 1 \tag{4.3h}$$

$$z_r \in \{0,1\}, \quad \forall r \in R \tag{4.3i}$$

$$y_r \in \{0,1\}, \quad \forall r \in R \tag{4.3j}$$

$$u_{j,a} \in \{0,1\} \quad \forall j \in RC(l) \cup LC(l), a \in S_j. \tag{4.3k}$$

Except for the first term, the objective function is the same as (4.2a). The new first term $\sum_{r \in R} W_r z_r$ computes the weighted sum of the number of rows being correctly classified. The weight W_r of a row r is generally set to 1. However, using the weights in the model allows us to train on the dataset where the rows have different weights (i.e., not all the rows are equally important). We also need this because our approach changes the weights of the rows in the dataset during the preprocessing stage.

Constraints (4.3b)-(4.3e) are the same as constraints (4.2b)-(4.2e). As previously mentioned, constraint (4.3e) for a row r is only useful if $\beta_r \geq 1$ for $r \in R_t$ and $\beta_r \geq 0$ for $r \notin R_t$ because of the objective direction. Constraints (4.3f)-(4.3g) ensure that the variable z_r takes value 1 if the row r reaches leaf l and has the same target as the path being generated. Constraint (4.3h) imposes the selection of exactly one target for the generated path.

Overall the new SP model (4.3) has $2 \times |R| + 1$ new constraints and $|R| + |T|$ new variables. Because of the merging, we have fewer SPs, but the model is larger than model (4.2).

4.3.2 Redundancy of MP Constraints (4.1c)

Observe that constraints (4.1c) in the integer MP (4.1) are dependent on the dataset but do not depend on the target of the rows. We show that these constraints are implied by constraints (4.1b), (4.1d)–(4.1f). The proof of this result relies on the the following lemma that is presented first.

Lemma 4.3.1. *In any solution to the model (4.1) that satisfies the constraints (4.1b), (4.1d), and (4.1f), there exists a split check $a_j^* \in S_j$ for every internal node $j \in N_{int}$ such that*

$$\rho_{j,a_j^*} = 1 \quad \textit{and} \quad \rho_{j,a} = 0, \quad \forall a \in S_j \setminus \{a_j^*\}. \tag{4.4}$$

Proof. Consider an internal node $j \in N_{int}$. There exists a leaf $l \in N_{lf}$ such that $j \in p_{BT}(l)$. We sum constraints (4.1d) over all candidate split checks $a \in S_j$ for leaf l and node j to obtain

$$\sum_{a \in S_j} \rho_{j,a} = \sum_{a \in S_j} \sum_{\substack{p \in DP_l: \\ s(j)=a}} x_p = \sum_{p \in DP_l} x_p = 1,$$

where the last equality follows from constraints (4.1b). Given that the $\rho_{j,a}$ variables are binary according to (4.1f), there is a single variable in the first sum, say for $a = a_j^*$, that takes value 1. All the others are equal to 0. □

Theorem 4.3.2. *For any decision tree with depth $k \geq 1$, constraints (4.1c) are implied by constraints (4.1b), (4.1d)–(4.1f). In other words, model (4.1) is equivalent to:*

$$Max \quad \sum_{l \in N_{lf}} \sum_{p \in DP_l} CP(p) x_p \tag{4.5a}$$

$$s.t. \quad \sum_{p \in DP_l} x_p = 1, \quad \forall l \in N_{lf} \tag{4.5b}$$

$$\sum_{\substack{p \in DP_l: \\ s_p(j)=a}} x_p = \rho_{j,a}, \quad \forall l \in N_{lf}, j \in p_{BT}(l) \cap N_{int}, a \in S_j \tag{4.5c}$$

$$x_p \in \{0,1\}, \quad \forall p \in DP_l, l \in N_{lf} \tag{4.5d}$$

$$\rho_{j,a} \in \{0,1\} \quad \forall j \in N_{int}, a \in S_j. \tag{4.5e}$$

Proof. Since model (4.5) is a relaxation of model (4.1), any feasible solution to model (4.1) is also feasible for model (4.5). We now show the opposite direction.

Consider an arbitrary data row $r \in R$. Let P_r be the subset of paths that are followed by row r from root node to some leaf node (which may have a different target than the row).

We need to show that for this row r, any feasible solution to model (4.5) contains exactly one path in set P_r.

Consider a feasible solution to model (4.5), which, therefore, satisfies constraints (4.1b), (4.1d), and (4.1f). By Lemma 4.3.1, we know that, in this solution, exactly one split check is assigned to each node. At the root node of the decision tree, let the assigned split check be $a = (f_a, \mu_a)$. Clearly, either $v_r^{f_a} \leq \mu_a$ and the row follows the left branch, or $v_r^{f_a} > \mu_a$ and the row follows the right branch. Using a similar argument at each other node reached by the row, we infer that row r visits a unique set of nodes (and split checks) to reach a unique leaf node $l \in N_{lf}$ in the tree.

In set P_r, there are exactly $|T|$ different paths that correspond to these sets of nodes and split checks, namely, one for each possible target in T. However, constraints (4.5b) and (4.5d) ensure that exactly one of these paths is selected for leaf l, implying that constraint (4.1c) for row r is satisfied by this solution of model (4.5). Consequently, all feasible solutions to model (4.5) are also feasible for model (4.1). □

4.3.3 Search for Violated MP Constraints (4.1c)

Although the constraints (4.1c) are implied in the integer MP (4.1), they are needed for a stronger linear relaxation (see results in Section 4.4.4). So, instead of removing them, we use them as cutting planes. In other words, we solve the RMP without constraints (4.1c) and add to the RMP only those that are violated by the linear relaxation solution. The search for the violated constraints is performed by inspection of the individual rows in R.

Furthermore, we experimented with considering only one bound (lower or upper) when adding constraints (4.1c) as cutting planes. We found that the upper bound inequality version of constraints (4.1c) performs similarly to using them as equality constraints. On the other hand, the lower bound inequality version of constraints (4.1c) results in a poor linear relaxation. Section 4.4.4 will detail experimental results.

The linear relaxation strength of model (4.1) depends on the dataset R. If it does not contain sufficient rows to obtain a strong bound, the column-generation-based heuristic may result in a poor-quality solution. Observe, however, that there is no need to limit the constraints (4.1c) to a predetermined dataset R. Indeed, any possible row described by a set of feature values should follow a single path in the tree to reach a leaf (independently of the target associated with this leaf). Consequently, we do not limit our search for violated constraints (4.1c) to the rows in set R, but also to any potential row for which we do not know the target. We will call the latter rows the *unlabeled rows* as no target will be associated with

them.

To generate an unlabeled row for which a given MP solution violates the corresponding constraint (4.1c), we develop a separation model. In fact, this model does not identify a specific row but determines whether the row would take the left or right branch on each split check. With this information, we can deduce bounds on the feature values that a row should have to yield a violated constraint and construct a corresponding unlabeled row by selecting arbitrary feature values respecting these bounds.

Consider the solution of the current RMP. Let DP denote the subset of paths p such that x_p takes a positive value x_p^* in this solution. Furthermore, for any row r, let P_r be the subset of paths in DP followed by this row. We want to generate an unlabeled row r (or equivalently, find the branch taken on each split check) such that $\sum_{p \in P_r} x_p^* \neq 1$. This constraint can be violated in two ways. Either the upper bound or the lower bound can be violated. From preliminary experiments using the existing constraints (4.1c) with only the lower or only the upper bound, we observed that using only the upper bound results in a stronger linear relaxation. Hence, we only focus on the violations of the upper bound.

Let θ_p be a binary variable that takes value 1 if the generated row follows path $p \in DP$. Let ψ_a be a binary variable that takes value 1 if the row takes the right branch on split check $a \in S$ and 0 otherwise. Thus, the row follows a path p if $\psi_a = 1$ for all split checks $a \in R_p$ and $\psi_a = 0$ for all split checks $a \in L_p$, where R_p (resp. L_p) is the set of split checks for which path p takes the right (resp. left) branch.

With these variable definitions, we can determine if the generated row follows a path $p \in DP$ using the following equation:

$$\theta_p = \bigwedge_{a \in L_p} \neg\psi_a \wedge \bigwedge_{a \in R_p} \psi_a. \tag{4.6}$$

As different split checks on possibly different paths may involve the same feature, there must be some consistency between the branches taken by the unlabeled row on these checks, i.e., there might be some implied relations between split checks. Assume that two split checks a and b are defined on the same feature f with different threshold values μ_a and μ_b with $\mu_a \leq \mu_b$. In that case, whenever the row takes the left branch on split check a ($v_r^f \leq \mu_a$ or, equivalently, $\psi_a = 0$), it must also take the left branch on split check b ($v_r^f \leq \mu_b$ or $\psi_b = 0$). This relation translates into the constraint

$$\neg\psi_a \implies \neg\psi_b. \tag{4.7}$$

Relying on the notation presented in Table 4.4, the separation model for generating violated

constraints (4.1c) for a given solution of the MP can be expressed as follows:

$$Max \quad \sum_{p \in DP} x_p^* \theta_p \tag{4.8a}$$

$$s.t. \quad \theta_p = \bigwedge_{a \in L_p} \neg\psi_a \wedge \bigwedge_{a \in R_p} \psi_a, \quad \forall p \in DP \tag{4.8b}$$

$$\neg\psi_a \implies \neg\psi_b, \quad \forall a, b \in S : f_a = f_b, \mu_a \leq \mu_b \tag{4.8c}$$

$$\theta_p \in \{0, 1\}, \quad \forall p \in DP \tag{4.8d}$$

$$\psi_a \in \{0, 1\}, \quad \forall a \in S. \tag{4.8e}$$

The objective function (4.8a) aims at finding an unlabeled row with a maximum left-hand side in constraint (4.1c). Constraints (4.8b) and (4.8c) have been introduced above.

Model (4.8) is solved by constraint programming, using the CP-SAT solver. A violated constraint (4.1c) is found whenever the optimal value of (4.8) is strictly greater than 1. In fact, we retrieve all intermediate solutions obtained by the solver that have an objective value greater than 1 and generate one cut for each of them. These cuts, if any, are then added to the MP.

4.4 Computational Results

In this section, we report the results obtained in our computational experiments. Section 4.4.1 gives an overview of the datasets used for these experiments and the experimental setup. In Section 4.4.2, we describe our preprocessing and initialization routines and show their effects. In Section 4.4.3, we report the computational results on merging the SPs as described in Section 4.3.1. In Section 4.4.4, we present the computational results for different ways of using constraints (4.1c) in the MP. Finally, in Section 4.4.5, we compare our revised column generation matheuristic with that of [1].

4.4.1 Datasets and Experimental Setup

We used 12 datasets from the UCI repository [70] for our computational experiments. There are six *small* datasets involving between 500 and 10,000 rows and six *large* datasets containing over 10,000 rows. The small datasets are also used in [43] and [1]. These datasets are already processed, and all features are numerical with no missing values. Each dataset is split into

Table 4.4 Notation for the separation model (4.8).

Sets	
DP	set of paths p with $x_p^* > 0$.
S	set of all split checks.
L_p	set of split checks where path p takes the left branch.
R_p	set of split checks where path p takes the right branch.
Parameters	
x_p^*	value of variable x_p in the current RMP solution.
f_a	feature of split check a.
μ_a	threshold of split check a.
Decision Variables	
θ_p	binary variable indicating if the generated row follows path $p \in DP$.
ψ_a	binary variable indicating if the generated row takes the right branch on split check $a \in S$.

training (50% of the rows) and testing (25%) parts, and the split is done randomly five times.[1] The reported performance for each dataset is averaged over these five train-test splits. We use the same train-test splits as [43] and [1].

For the large datasets, the train-test split used by [1] is not available. We performed the same cleaning steps as in [1], including transforming classes to integers and converting string features into numerical ones. For each dataset, we then generated five random train-test splits like for the small datasets. The dataset specifications are listed in Table 4.5. The datasets and the source code are available at https://github.com/krooonal/col_gen_estimator/tree/dtreedev.

For the experiments, we used a cluster of CPU servers, each with two sockets of Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz, 28 cores each (total of 56 cores), and 512GB RAM. However, for training, each process was limited to 8 threads and 16 GB of memory. We imposed these limits to ensure a fair comparison with [1]. For the comparison, we use the numbers reported in [1].

We used Python 3 for all computations, scikit-learn version 1.2.2 [19] for running CART, Gurobi 9.5.0 [67] to solve the RMP (with or without integrality requirements), and the CP-SAT solver from Google OR-Tools [66] to solve the SPs ((4.2) or (4.3)) and to generate cutting planes by solving (4.8). Indeed, preliminary experiments showed that using the CP-SAT solver for the SPs is about twice as fast as using Gurobi. Since the CP-SAT solver can only solve problems with integer coefficients, we multiply the objective coefficients by a

[1]The remaining 25% was used by [44] for validation but, like [1], we did not use it.

Table 4.5 Dataset sizes.

Dataset	Number of rows	Number of features	Number of classes
Small			
Tic tac toe	958	18	2
Indian diabetes	768	8	2
Car evaluation	1728	5	4
Seismic bumps	2584	18	2
Spambase	4601	57	2
Statlog satellite	4435	36	6
Large			
Default Credit	30000	23	2
Hand posture	78095	33	5
HTRU 2	17898	8	2
Letter Recognition	20000	16	26
Magic04	19020	10	2
Shuttle	43500	9	7

scaling factor 10^5 and round them to the nearest integer. This is equivalent to solving the model using a MIP solver with 10^{-5} as the optimality threshold.

Our computational approach is the same as in [1] except for the optional preprocessing and initialization steps 4 and 5 below. The preprocessing step is further described in Section 4.4.2. The complete process is as follows:

1. Run 300 iterations of CART on randomly selected 90% of the training data and collect split checks for each node.

2. Select the q most frequent split checks for each internal node as the candidate split checks, where $q = \left\lfloor \frac{150}{|N_{int}|} \right\rfloor$ for the root node and $q = \left\lfloor \frac{100}{|N_{int}|} \right\rfloor$ for the other nodes.

3. Run CART on the entire training data. Add the generated split checks to the candidate split checks and the generated paths to the initial RMP. This step ensures that the final output tree will have accuracy at least as high as the CART output.

4. [Optional] Run 100 iterations of CART on randomly selected 80% of the training data and add the generated paths to the initial RMP. A path is added only if the split checks it contains belong to the candidate split checks of the corresponding nodes.

5. [Optional] Preprocessing: If any two rows in the dataset have the same target and the rows take the same branches on all candidate split checks of the reachable nodes, remove one of the rows and increase the weight of the other row by one.

6. Perform column generation iterations to solve the MP. Stop if it is solved to optimality (all SPs failed to generate new columns) or a time limit is reached.

7. Solve the integer MP (4.1) restricted to the generated columns to get the final decision tree.

In our revised column generation approach, we apply a simpler version of the pricing heuristic presented in [1]. We start with randomly selecting a leaf $l \in N_{lf}$, then for each node $j \in p_{BT}(l)$, we randomly select a split check $a \in S_j$ that has not been selected before. If we are able to select different split checks for each node in the path, we compute the target of the path that maximizes the accuracy for the set of rows following this path. Finally, we compute the reduced cost of the generated path using the duals from model (4.1). If the reduced cost is greater than the threshold value 10^{-6}, we add the path to the RMP. We repeat this process 100 times in each column generation iteration. Unlike [1], we do not maintain a set of feasible columns that are not yet added to the RMP.

Except for the experiments in Section 4.4.5, the depth of the tree is fixed to $k = 4$ for all our experiments.

4.4.2 Initialization and Preprocessing Results

In steps 1–3 above, we follow the process of generating candidate split checks as in [1]. To initialize the RMP, [1] only retain the paths generated from the last run of CART on the entire dataset (step 3). We extend further this initialization by collecting all paths generated during additional iterations of CART on a randomly selected subset of the training dataset (step 4).

After generating all the candidate split checks and paths for initialization, we compute the set of reachable nodes for each row in the training dataset. A node j is reachable by a row r if an assignment of split checks exists for the ancestor nodes of node j such that row r can reach node j by following the branches of the ancestor nodes. We use these sets of reachable nodes to compare pairs of rows in the training dataset.

If two rows take the same branch on all the split checks on the reachable nodes, for one of the rows, we remove the corresponding constraint (4.1c) from integer MP (4.1) and constraints (4.3c)-(4.3e) from the SP (4.3) (or constraints (4.2c)-(4.2e) from the SP (4.2)). We then add 1 to the weight W_r of the other row r that is kept as it is. Removing such duplicate constraints reduce the size of integer MP (4.1) and SPs (4.3) (or SPs (4.2)).

We trained the models with three variations of our approach to evaluate the effects of the

preprocessing step 5 and the initialization step 4. For these experiments, we imposed no time limit. The variations are the following:

Default: Default training process with steps 4 and 5 enabled.

No Preprocess: Training process with preprocessing in step 5 disabled.

No Init: Training process with extra initialization in step 4 disabled.

Figure 4.1 shows the solving time for all variations.[2] We can observe that *No Preprocess* gives the best solving times for the small datasets (left part of the black divider) except for the Car evaluation dataset. However, for large datasets, it gives the worst solving times except for Magic04 dataset. This suggests that the preprocessing step 5 is helpful only on large instances.

The solving times for the *No Init* variant are always larger than the *Default* variant except for Spambase and Shuttle datasets. This suggests that extra initialization is helpful in general.

All variations have similar accuracy gains over CART, as expected (see Figure 4.2). The average training accuracy gain over CART is 0.9% for all variations.

4.4.3 Merged SPs Results

To evaluate the effect of merging the SPs, we disabled the pricing heuristic. We set a time limit of 1 hour (without the pricing heuristic, the solving times are too large). The extra initialization and preprocessing steps 4 and 5 were enabled for all datasets.

The merged SPs are generally faster than the original SPs and hence add more columns to the RMP in the same time limit. We can observe this in Figure 4.3. Furthermore, as shown in Figure 4.4, the extra columns added because of the faster SPs lead to larger gains in accuracy over CART on the training dataset. The dataset 'Tic-tac-toe' is the only notable exception where the original SPs added more columns compared to the merged SPs. However, the accuracy gains over CART are larger for the approach with the merged SPs.

4.4.4 MP Constraints (4.1c) Results

To evaluate the effect of constraints (4.1c), also referred to in the following as the *beta cuts*, we experimented with the following six algorithm variants:

[2]Result for the Letter Recognition instances is not included as the training process did not finish after 15 hours.

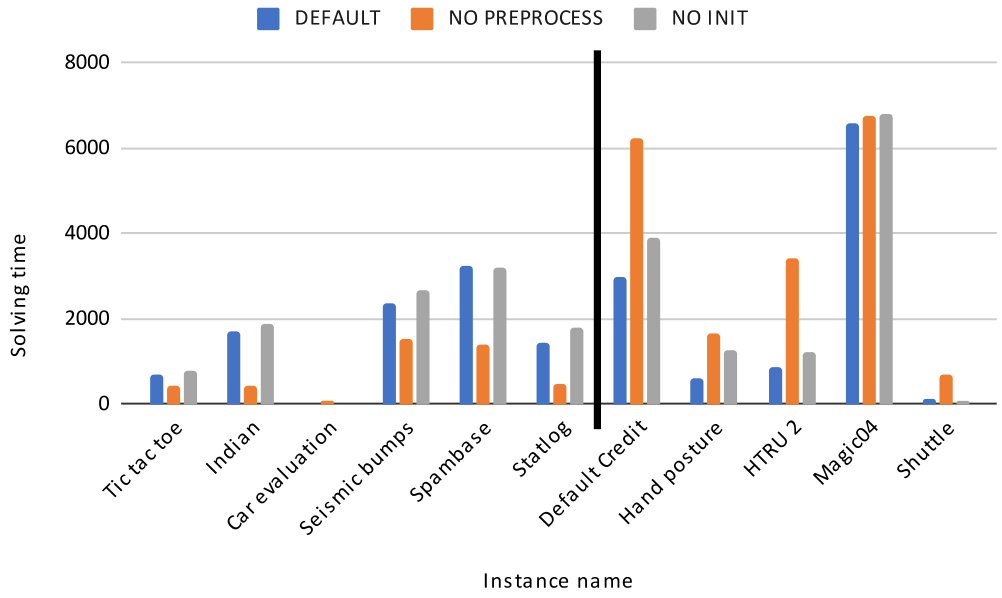

Figure 4.1 Solving time for different ways of using preprocessing and initialization.

No Beta: The MP is solved without using any constraints (4.1c).

Beta Lb: The MP does not contain any constraints (4.1c) at the beginning, but they are added as inequality cuts with only the lower bound.

Beta Ub: The MP does not contain any constraints (4.1c) at the beginning, but they are added as inequality cuts with only the upper bound.

Beta Eq: The MP does not contain any constraints (4.1c) at the beginning, but they are added as equality cuts.

All Beta: The MP contains all constraints (4.1c) from the beginning. This is the setting used by [1].

Extra Beta: Same as the *Beta Eq* variant except that additional equality cuts for unlabeled rows can be generated using model (4.8).

Consequently, beta cuts associated with unlabeled rows are only generated in the *Extra Beta* variant. In all variants with cuts, the cut generation algorithm is invoked at every 10 iterations of column generation. For these experiments, we considered no time limit and enabled the initialization and preprocessing steps 4 and 5.

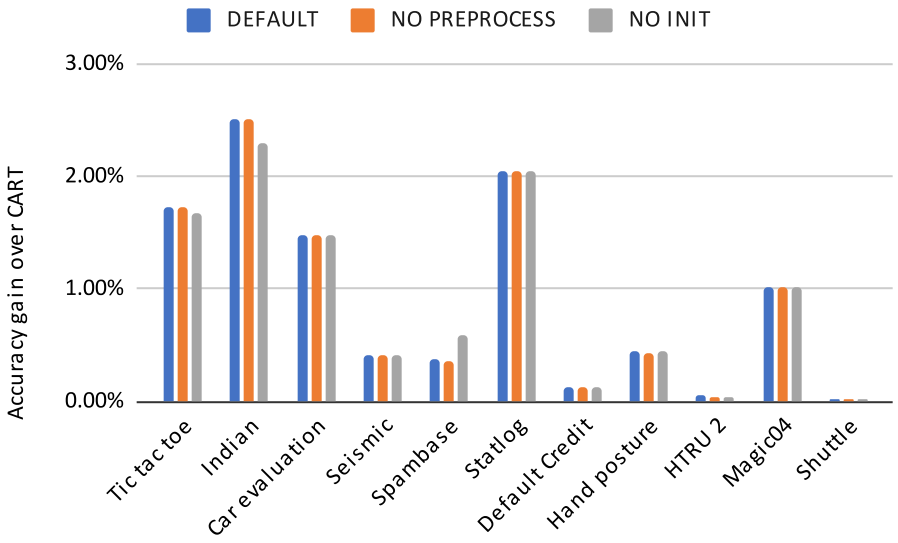

Figure 4.2 Accuracy gain over CART for different ways of using preprocessing and initialization.

Figure 4.5 shows the solving times for all variants.[3] The solving times for *No Beta* are the shortest among all variants. The *Beta Lb* variant has similar (but slightly larger) solving times than *No Beta*. However, these two variants provide the worst linear relaxation value compared to the others (see MP Value in Figure 4.6). Hence, they also result in the lowest accuracy gains over CART compared to the other variants (see Figure 4.7).

The solving times for *Beta Ub* are larger than those of *No Beta*. The *Beta Eq* variant has similar but slightly larger solving times compared to the *Beta Ub* variant. These two variants are consistently faster than the *All Beta* variant. Finally, the variant *Extra Beta* has the largest solving times. These four variants (*Beta Ub*, *Beta Eq*, *All Beta*, and *Extra Beta*) have similar linear relaxation (Figure 4.6) and result in similar accuracy gains over CART on the training dataset (Figure 4.7). The *Extra Beta* variant has slightly better accuracy gains over CART compared to the other three variants. However, the difference is too small to observe it in the diagrams (on average 0.95% versus 0.93% for *Beta Ub*, *Beta Eq*, and *All Beta* variants).

Table 4.6 shows the average training accuracy gains over CART when we use a time limit of 600 seconds. The gains for *Beta Ub*, *Beta Eq*, *All Beta*, and *Extra Beta* are comparable and much better than those achieved by the other two variants. Given that the average solving time of *Beta Ub* is less than the three others, we conclude that *Beta Ub* has the best trade-off

[3]Results for the Letter Recognition dataset are not included as the training process did not finish after 15 hours.

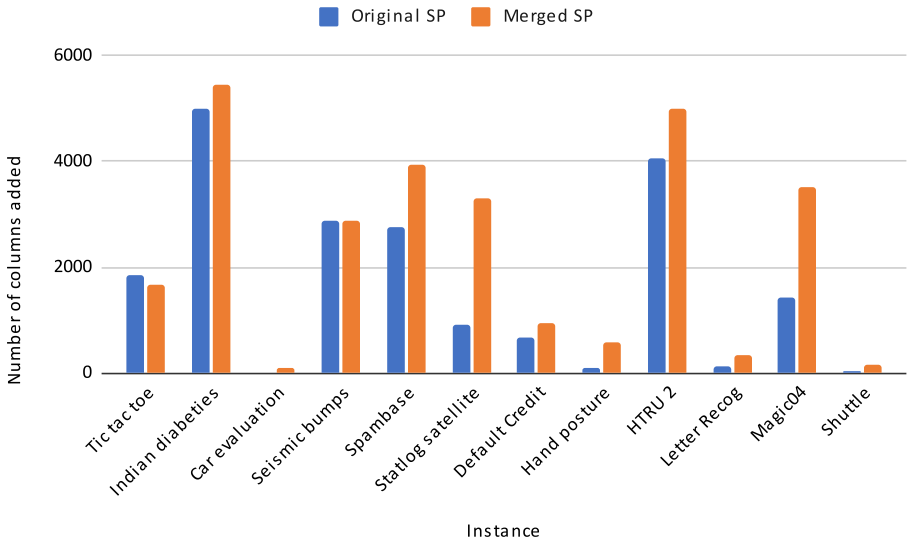

Figure 4.3 Number of columns added with original and merged SPs.

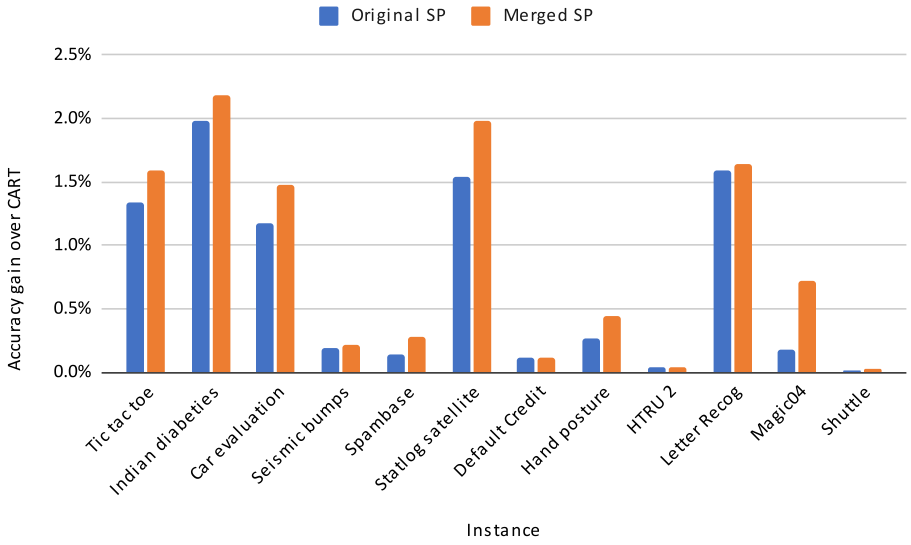

Figure 4.4 Accuracy gain over CART on training datasets with original and merged SPs.

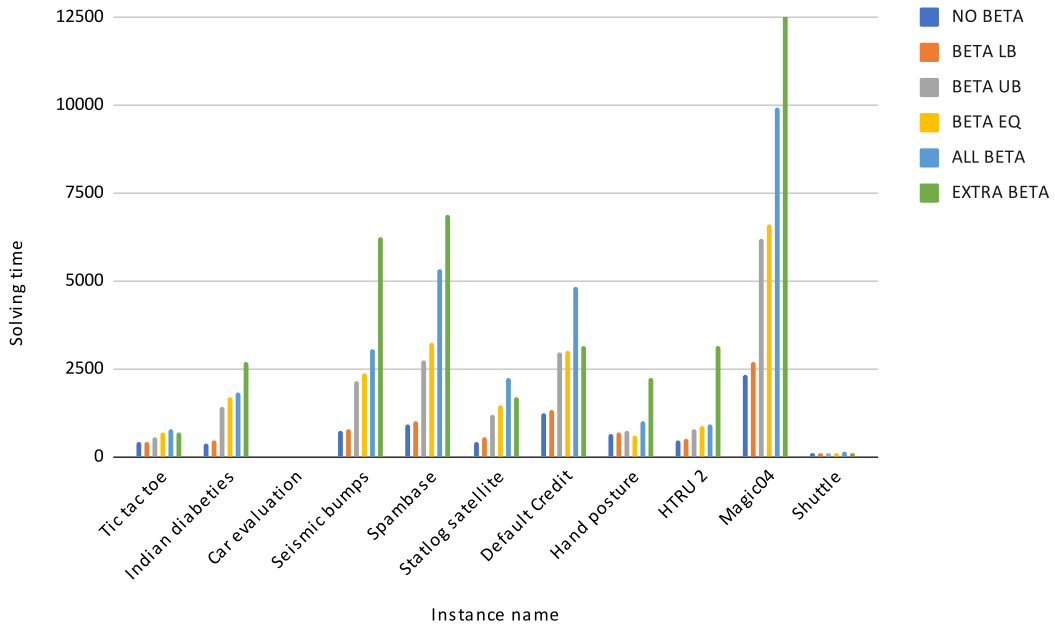

Figure 4.5 Solving time for different ways of using constraints (4.1c).

between solving time and training accuracy gain over CART.

Table 4.6 Average training accuracy gain over CART for different ways of using constraints (4.1c) with a 600s time limit.

Method	Train accuracy (%)	Gain over CART (%)
CART	81.40	0.00
No Beta	81.85	0.45
Beta Lb	82.04	0.64
Beta Ub	82.29	0.90
Beta Eq	82.29	0.89
All Beta	82.28	0.87
Extra Beta	82.26	0.86

Our approach does not consider all possible split checks for each node. Hence, we cannot prove the optimality of the generated tree even if we would be applying a full branch-and-price algorithm. However, for the problem limited to the candidate split checks for each internal node, we can analyze the optimality of our solutions. We compared the MP optimal values to the values of the best integer solutions found. The *Extra Beta* variant can prove

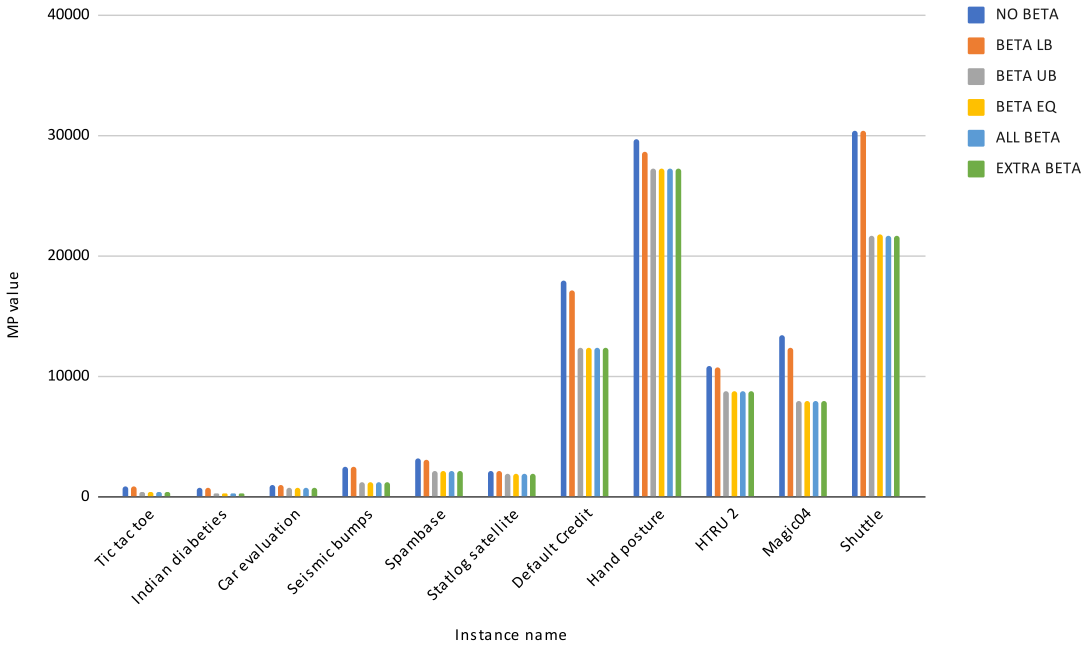

Figure 4.6 MP value for different ways of using constraints (4.1c).

optimality for 53 instances out of 55 instances (see Figure 4.8). Thus, in most cases, we can prove optimality using a heuristic approach. This suggests that branch-and-price might not improve solution quality significantly in the presence of the beta cuts generated using model (4.8). Note that the solution values produced by the other variants (*Beta Ub*, *Beta Eq*, and *All Beta*) are very close to the optimal values, as shown in Figure 4.7.

4.4.5 Comparison with [1]

This section compares our best results against the results reported in [1]. For these experiments, we used the merged SP model (4.3). We disabled the preprocessing step 5 for small datasets and enabled it for the large datasets, while the initialization step 4 was enabled for all datasets. As done by [1], we set a time limit of 600 seconds for the training and considered three tree depths $k = 2, 3, 4$.

Table 4.7 compares training accuracy gains over CART against the column generation approach of [1] where CG stands for column generation. Better gains are highlighted in bold. The revised column generation approach achieved strictly larger gains on 13 cases and strictly lower gains on 2 cases out of a total of 18 comparisons.

The accuracies for training are not available for the large datasets in [1]. We compare the

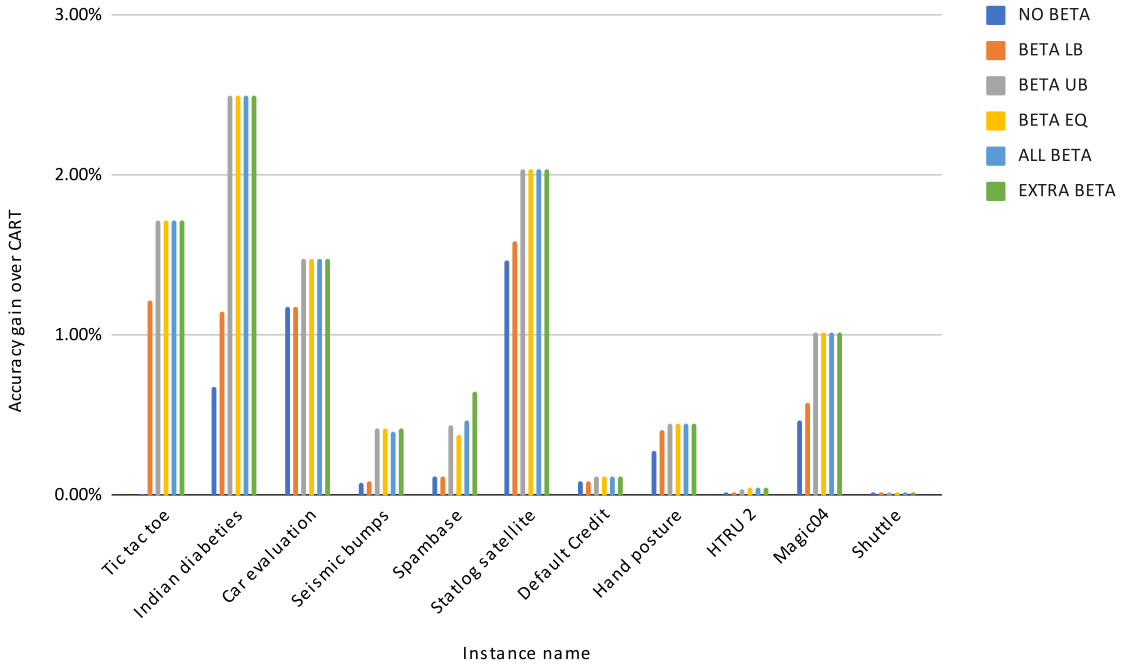

Figure 4.7 Training accuracy gain over CART for different ways of using constraints (4.1c).

testing accuracy gains against the column generation approach of [1] in Table 4.8. The revised column generation approach achieves strictly larger gains on 6 cases and strictly lower gains on 3 cases out of 18 instances. We also report the training accuracy gains obtained by our column generation heuristic for the large datasets in Table 4.9. Out of 18 instances, our heuristic yields a gain on 11 instances, with a maximum gain of 3.2%.

Note that both approaches focus on improving the training accuracies and do not take extra steps to generalize the performance across the testing datasets. However, even if the model is not trained having generalization in mind, it still can significantly improve over CART. With more focus on generalizing the performance over unseen data, we should be able to produce even better results. We discuss some directions to address this issue in the next section.

4.5 Conclusion and Future Work

In this work, we presented modifications to the column-generation-based heuristic of [1] that can be applied to generate decision trees. First, we reduced the number of SPs by moving the target computation in the constraints. This results in a faster generation of columns. Then, we showed that the data-dependent constraints in the integer MP are implied and

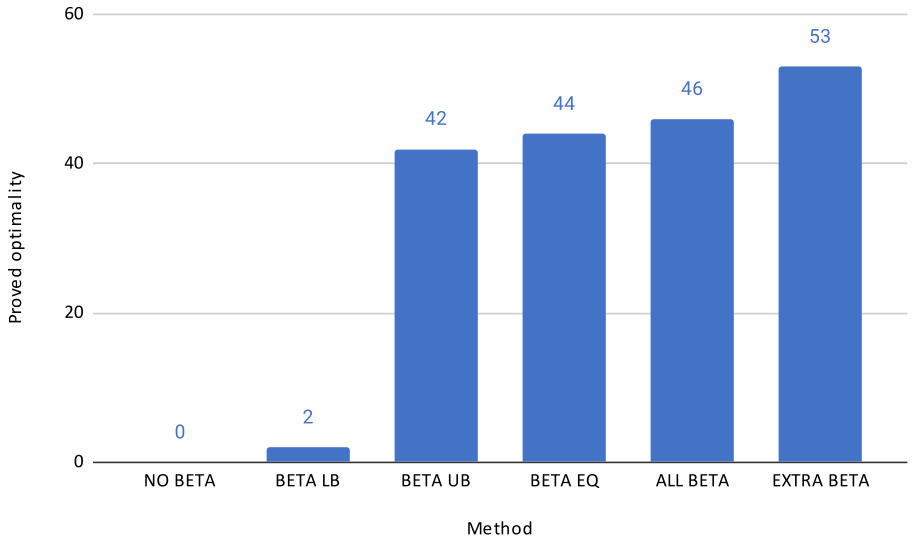

Figure 4.8 Number of instances solved to optimality for different ways of using constraints (4.1c).

presented ways to use them as cutting planes. This helps solve the RMP faster. Furthermore, we described an optimization model to generate these cutting planes on demand even if the corresponding data row is not in the training dataset. These additional cutting planes helped to show that we can generate optimal decision trees for the given candidate split checks in most instances. Note that this model can also be linearized into a MIP model, an avenue that might be explored in future research. Finally, we described a process for better initializing the RMP and preprocessing the dataset to reduce the size of the model. The extra initialization and preprocessing steps further help to reduce the solving times, especially for the larger datasets.

As future work, we can consider developing a diving heuristic (see [69]) to derive better integer solutions for the variant where constraints (4.1c) are not used (*No Beta*). This variant has the best solving time but the worst solutions compared to the other variants. Also, as the SPs only change in the objective function across the column generation iterations, we can explore how the information generated by solving previous SPs can be exploited to solve the subsequent SPs faster [71].

Finally, as we did not focus on generalizing the performance of our approach for out-of-sample datasets, the suggestions made by [1] to generalize this performance can be studied, namely, to penalize the number of active leaves (i.e., reached by at least one row) in the integer MP and to enforce in the SP a minimum number of rows following any generated path. Like

Table 4.7 Comparision of training accuracy gains for small datasets.

		Accuracy (%)					
		Firat et al. CG			Revised CG		
Instance	k	CART	CG	Gain	CART	CG	Gain
Tic tac toe	2	71.2	71.8	**0.6**	71.3	71.8	0.5
	3	75.4	76.7	1.3	75.4	77.4	**1.9**
	4	84.4	85.4	1.0	84.5	86.2	**1.8**
Indian diabeties	2	77.3	78.8	**1.5**	77.3	78.8	**1.5**
	3	78.9	81.2	2.3	78.9	81.3	**2.4**
	4	82.9	84.2	1.3	82.9	85.3	**2.4**
Car evaluation	2	76.9	76.9	**0.0**	76.9	76.9	**0.0**
	3	79.0	79.8	0.8	79.0	80.2	**1.2**
	4	84.2	85.2	1.0	84.2	85.7	**1.5**
Seismic bumps	2	93.1	93.3	0.2	93.1	93.4	**0.3**
	3	93.4	93.7	**0.3**	93.4	93.7	**0.3**
	4	93.9	94.2	0.3	93.9	94.3	**0.4**
Spambase	2	86.0	87.1	1.1	85.9	87.2	**1.3**
	3	89.6	90.3	**0.7**	89.6	90.3	0.6
	4	91.6	91.6	0.0	91.6	92.0	**0.4**
Statlog satellite	2	63.2	64.0	0.8	63.2	64.3	**1.1**
	3	78.7	79.5	0.8	78.7	80.0	**1.3**
	4	81.6	82.9	1.3	81.6	83.6	**2.0**

minimizing the number of internal nodes in [44], minimizing the number of active leaves aims at reducing tree complexity. It would be interesting to study the performance of the proposed heuristic with these changes.

Table 4.8 Comparision of testing accuracy gains for the large datasets.

		Accuracy (%)					
		Firat et al. CG			Revised CG		
Instance	k	CART	CG	Gain	CART	CG	Gain
Default Credit	2	82.3	82.3	**0.0**	81.9	81.9	**0.0**
	3	82.3	82.3	**0.0**	82.0	82.0	**0.0**
	4	82.3	82.3	**0.0**	81.9	82.0	**0.0**
Hand posture	2	56.4	56.4	**0.0**	56.6	56.6	**0.0**
	3	62.5	62.8	0.3	62.6	63.0	**0.4**
	4	69.0	69.1	0.1	69.4	69.7	**0.3**
HTRU 2	2	97.8	97.8	**0.0**	97.6	97.6	**0.0**
	3	97.9	97.9	**0.0**	97.7	97.7	**0.0**
	4	98.0	98.0	**0.0**	97.8	97.7	-0.1
Letter Recog	2	12.5	12.7	0.2	12.3	12.7	**0.4**
	3	17.7	18.6	0.9	17.6	19.6	**2.0**
	4	24.8	27.0	2.2	24.5	27.8	**3.3**
Magic04	2	78.4	79.1	**0.7**	79.0	79.7	**0.7**
	3	79.1	80.1	**1.0**	79.1	79.9	0.8
	4	81.5	81.5	0.0	81.6	82.4	**0.8**
Shuttle	2	93.7	93.7	**0.0**	93.9	93.9	**0.0**
	3	99.6	99.7	**0.1**	99.6	99.7	0.0
	4	99.8	99.8	**0.0**	99.8	99.8	**0.0**

Table 4.9 Training accuracy gains of the revised CG heuristic for the large datasets.

		Accuracy (%)		
Instance	k	CART	CG	Gain
Default Credit	2	82.0	82.0	0.0
	3	82.2	82.2	0.0
	4	82.3	82.5	0.1
Hand posture	2	56.6	56.6	0.0
	3	62.7	63.1	0.5
	4	69.4	69.9	0.4
HTRU 2	2	97.8	97.8	0.1
	3	98.0	98.1	0.1
	4	98.2	98.3	0.0
Letter Recog	2	13.1	13.4	0.3
	3	18.2	20.8	2.6
	4	25.6	28.8	3.2
Magic04	2	79.6	80.0	0.4
	3	80.0	81.0	1.0
	4	82.7	83.7	1.0
Shuttle	2	93.9	93.9	0.0
	3	99.7	99.7	0.0
	4	99.8	99.9	0.0

CHAPTER 5 A FRAMEWORK FOR COLUMN-GENERATION-BASED CLASSIFIERS IN SCIKIT-LEARN

5.1 Introduction

Scikit-learn [19] is a free and open-sourced software ML library for the Python programming language. Its goal is to make ML tools available to ML experts and beginners. It includes different ML models for classification, regression, and clustering tasks, e.g., support vector machines, decision trees, random forests, gradient boosting, k-means, and dimensionality reduction. Apart from the ML models, it also includes tools for developing end-to-end pipelines for ML tasks. That includes tools for feature extraction and normalization, splitting the data, and tuning the hyperparameters.

Scikit-learn is widely adopted in industry and academia due to its user-friendly design and rich tools and algorithms. Scikit-learn's design philosophy emphasizes consistency in its API, making it easier for users to switch between different ML algorithms seamlessly [65]. The active and diverse community surrounding scikit-learn ensures that the library remains up-to-date, with frequent updates and contributions from experts in the field, making it a reliable choice for ML tasks.

Scikit-learn's flexibility and widespread adaptability motivate the integration of CG-based classifiers discussed in the previous chapters into scikit-learn. In this chapter, we describe a software framework that makes it easier for developers to design and implement CG-based ML models on top of scikit-learn. The main goal is to provide an efficient, flexible, well-documented implementation of the common components used across various CG-based classifiers.

We implemented the CG framework using a scikit-learn classifier template [72]. This framework is a template for building CG-based classifiers compatible with scikit-learn. The implementation has been tested through unit tests and examples. The framework code follows PEP 8, a consistent style guide recommended by Rossum et al. [73]. The framework code is well documented.[1] The framework code is open-sourced and accessible at https://github.com/krooonal/col_gen_estimator.

Further analysing the classifiers developed in Chapters 3 and 4, we noticed that over 96% of the training time is spent in solving MIP subproblems. This motivates the need to solve

[1]The documentation for various classes in the framework can be accessed at https://col-gen-estimator.readthedocs.io/en/latest/api.html.

the subproblems faster. In both of these classifiers, the subproblems are MIPs. During CG iterations, these subproblems are solved in a sequential manner and only differ in the objective function. By exploiting the similarities in the subproblems and reusing the information generated by solving subproblems in previous iterations, we can reduce the training times for these classifiers. Because of the growing need of solving similar MIPs sequentially in many other applications, the MIP 2023 workshop organized a computational competition [74] that focused on reoptimization of MIPs. In Chapter 6, we present our approach that was awarded the first prize in this competition. We use some of the techniques developed for the competition in our CG-framework and plan to include others in the future.

This chapter is organized as follows. Section 5.2 presents the design of the framework classes and APIs. Section 5.3 describes how a CG-based classifier works using this framework. Section 5.4 presents two examples of CG-based classifiers built using this framework. Finally, Section 5.5 shows our approach to exploit similarities in the subproblem to improve the solving times.

5.2 Framework Design

The CG framework includes abstract classes for the MP (Section 5.2.1), subproblem (Section 5.2.2), and the classifier (Section 5.2.3). Since most CG-based ML models do not use branch-and-price, the current version implements the restricted master heuristic to solve the integer MP. For this section, we denote our training data with X and training labels as y.

5.2.1 Master Problem

The framework implements an abstract class for the MP called *BaseMasterProblem*. This class stores the RMP, current primal, and current dual solutions. This class also stores a Google OR-Tools MPSolver object in the 'solver_' variable. The developer can load the MP model in the 'solver_' variable to facilitate the communication with other methods in the class. The developer can also choose not to use it. However, it is recommended that the developer creates the MP using the provided MPSolver object.

This class contains the following abstract methods that need to be extended by the developer:

- **___init___(hyperparameters):** Creates the class object. This method records hyperparameters for the MP in the class.
- **generate_mp(X,y):** Creates and stores the RMP model using the given training data. It also records which variables are integer. It stores the generated constraints

and variables for retrieving primal and dual solutions.

- **add_column(column):** Adds the given column (generated by some subproblem) to the RMP.
- **solve_rmp(solver_parameters):** Solves the RMP using the given solver parameters. The method records the primal solution and returns the dual costs of constraints and other relevant information (for example, in case of integer MP, information about generated cuts in the RMP). This information is passed to the subproblems to generate new columns.
- **solve_ip(solver_parameters):** Solves the integer RMP (if the MP is a MIP) using the given solver parameters. The method records the primal solution in the class. The classifier uses this solution to generate predictions on training and testing data.

5.2.2 Subproblem

The subproblem generates new columns for a given dual solution of the RMP. In order to generate columns, the subproblem may solve an LP/MIP or use other methods, including heuristic approaches. A CG-based classifier may have multiple subproblems. For example, to solve model (4.3) we use only one subproblem for each leaf node in the tree.

This class also stores a Google OR-Tools MPSolver object in the 'solver_' variable and Google OR-Tools CP-SAT solver in the 'cp_solver_' variable. The developer can choose not to use them if they are not needed to generate columns.

This class contains the following methods that need to be extended by the developer:

- **___init___(hyperparameters):** Creates the class object. This method records hyperparameters for the subproblem in the class.
- **update_subproblem($\boldsymbol{X}$,$\boldsymbol{y}$, dual_solution):** Generates or updates the previously generated subproblem model. This method is optional since not all subproblems need to store a mathematical model. The base class implementation of this method does nothing.
- **generate_columns($\boldsymbol{X}$,$\boldsymbol{y}$, dual_solution, parameters):** Solves the subproblem and returns zero or more new columns. The developer may use the 'solve_model_' method implemented in the base class to use the CP-SAT solver with automated parameter tuning.

5.2.3 Classifier

This class adds an abstract classifier in Scikit-learn that uses CG. The reader is referred to [72] for a detailed description of the scikit-learn classifier template. It inherits the *Estimator* and the *Predictor* interfaces of scikit-learn and contains the following methods:

- **___init___(hyperparameters):** Creates the class object. This method creates the MP (implemented using a derived class of BaseMasterProblem), subproblem(s) (implemented using derived class(es) of BaseSubproblem), and records the hyperparameters for the classifier. The hyperparameters determine when the CG loop terminates and whether the integer RMP must be solved.
- **fit($\boldsymbol{X}$,$\boldsymbol{y}$):** This method is responsible for training the classifier by implementing the CG loop. It calls the corresponding methods for generating the MP, solving the RMP, generating columns, and adding the generated columns to the RMP. The CG loop can also be terminated early, for example, if the iteration limit is reached or there has been no significant improvement in the last few iterations. These conditions can be specified using hyperparameters. This method is not abstract. The derived class should not change any part of this.
- **predict($\boldsymbol{X}$):** Uses the solution values in the MP and produces a prediction. The developer needs to extend this abstract method and provide a logic for predicting the output.
- **predict_proba($\boldsymbol{X}$):** Similar to predict method. This method returns the probability for a given prediction. Implementing this method is optional.

The classifier stores the subproblems in multiple levels. The level-based implementation allows us to record priorities for subproblems. The lower-level subproblems have higher priorities. The classifier calls a subproblem at level $i > 1$ only if the subproblems at level $i-1$ fail to generate any columns. Hence, we can use lower-level subproblems for implementing fast heuristic algorithms and higher-level subproblems for slower mathematical models to generate columns. We can implement multiple subproblems at a level. In the current classifier implementation, the classifier calls the subproblems at the same level in sequence. However, in the future, we plan to call the subproblems at the same level in parallel.

The classifier also computes and stores the following statistics that are often needed for performance analysis:

- **performed_iter_:** Total number of iterations performed in column generation loop.

- **num_improving_iter_:** Total number of iterations where the master problem objective was improved.
- **mp_optimal_:** Set to true if none of the subproblems could generate any column in an iteration.
- **num_col_added_sp_:** Count of number of columns added to the master problem by each subproblem.
- **time_spent_sp_:** Total time spent by each subproblem.
- **time_spent_master_ :** Total time spent by the master problem.
- **time_add_col_ :** Total time spent in adding columns to the master problem.
- **time_limit_reached_ :** Set to true if the given time limit is reached.

5.3 Workflow

The workflow of a CG-based classifier in this framework involves three stages: Initialization, Training, and Prediction.

5.3.1 Initialization

In the initialization stage, the classifier object is created. This is done using the constructor (*___init___* method) described above.

This method starts with recording the hyperparameters of the classifier. Some hyperparameters are common for all CG-based classifiers, such as limits on iterations and time. These hyperparameters are passed to the constructor of the base class of the classifier. Other hyperparameters are specific to the CG-based classifier under construction; for example, a decision tree classifier's hyperparameters include the tree's depth and paths used for initialization. Such hyperparameters are recorded in the derived class of the classifier.

A classifier may have certain assumptions regarding hyperparameters for it to work correctly. For example, in a decision tree, a path of length 7 used for initialization is invalid for a tree of depth 4. Recording the hyperparameters also includes optional validation steps where such assumptions are checked.

The constructor then proceeds to create an MP object. Note that the MP optimization model is not yet created since it depends on the training data. The optimization model is populated in the MP object during the training stage.

Finally, the constructor creates the subproblem object(s). Like the MP, the constructor only creates the objects for the subproblem classes. The optimization models are created during the training stage. Subproblem heuristic objects are stored at lower levels, and the subproblem optimization models that require more solving time are stored at higher levels, as described above.

5.3.2 Training

In the training stage, the classifier is trained using the given training data with the *fit* method described above. It begins with validating the given training data by calling a method provided by scikit-learn: *check_X_y*. This method checks the given training data and labels for consistent dimensions and ensures that each training data has a non-empty label. The classifier then transforms the labels using a label encoder. It converts the labels into a sequence of integers starting from 0. This transformation helps with the creation of optimization models used in the classifier.

The *fit* method then generates the MP model by calling the *generate_mp* method with the given training data. The subproblems are created during the first iteration of the CG loop.

The *fit* method then initializes some statistics, including iteration number, total time spent in the MP, total time spent in the subproblems, and number of columns added by each subproblem. These statistics can be used to analyze the performance of different CG-related components.

The next step during the training stage involves the CG loop. This involves calling the following methods:

- *solve_rmp* in the MP to solve the RMP and return the dual values.
- *update_subproblem* for each subproblem. This either creates the subproblem model or updates its objective using the duals received from MP.
- *generate_columns* for subproblems if the previous subproblems failed to generate any columns. The loop terminates if all subproblems fail to generate columns.
- *add_column* for each column generated by subproblems.

The loop terminates if the iteration limit or time limit is reached or if no column is returned by the subproblem(s). If the MP has integer variables, then the *solve_ip* method is called to solve the integer MP using the restricted master heuristic.

5.3.3 Prediction

During the prediction stage, the classifier receives data points and predicts the labels (or probabilities for the labels) for them. This is done using the *predict* or the *predict_proba* methods of the classifier.

The prediction stage starts with validating the given data by calling the *check_array* method provided by scikit-learn. It ensures that the given data has finite values. The *predict* method also ensures that the classifier is trained (i.e., the *fit* method has been called at least once).

The next step in the prediction stage involves computing labels (or the probabilities for the labels) for the data points. This requires using the solution stored in the MP during the training stage.

Finally, in the *predict* method, the computed labels are transformed back to the original labels using the inverse transformation of the label encoder used during the training stage.

Figure 5.1 shows the interaction among different classes in the workflow of the CG Framework. The numbered text on each arrow indicates the order of operations. The yellow parts represent the Initialization stage. The green parts represent the Training stage, with light green representing the CG loop. The red part represents the Prediction stage.

5.4 Examples

This section describes two examples of CG-based model classifiers that use the framework described above.

5.4.1 Boolean Decision Rule Generation

Section 3.2 describes the boolean decision rule generation classifier that generates boolean decision rules for binary classification using CG.

The MP implements the methods *generate_mp, add_column, solve_rmp* and *solve_ip* using model (3.1). The MP uses the hyperparameter C (maximum complexity of the rule).

For this classifier, the subproblems are implemented in two levels. In level 0, we implement the heuristic to generate clauses quickly. This subproblem only implements the *generate_columns*. The *update_subproblem* remains an empty method. In level 1, we implement the subproblem model (3.2). This uses the hyperparameter D (maximum length of the clause). This subproblem implements both *generate_columns* and *update_subproblem* methods.

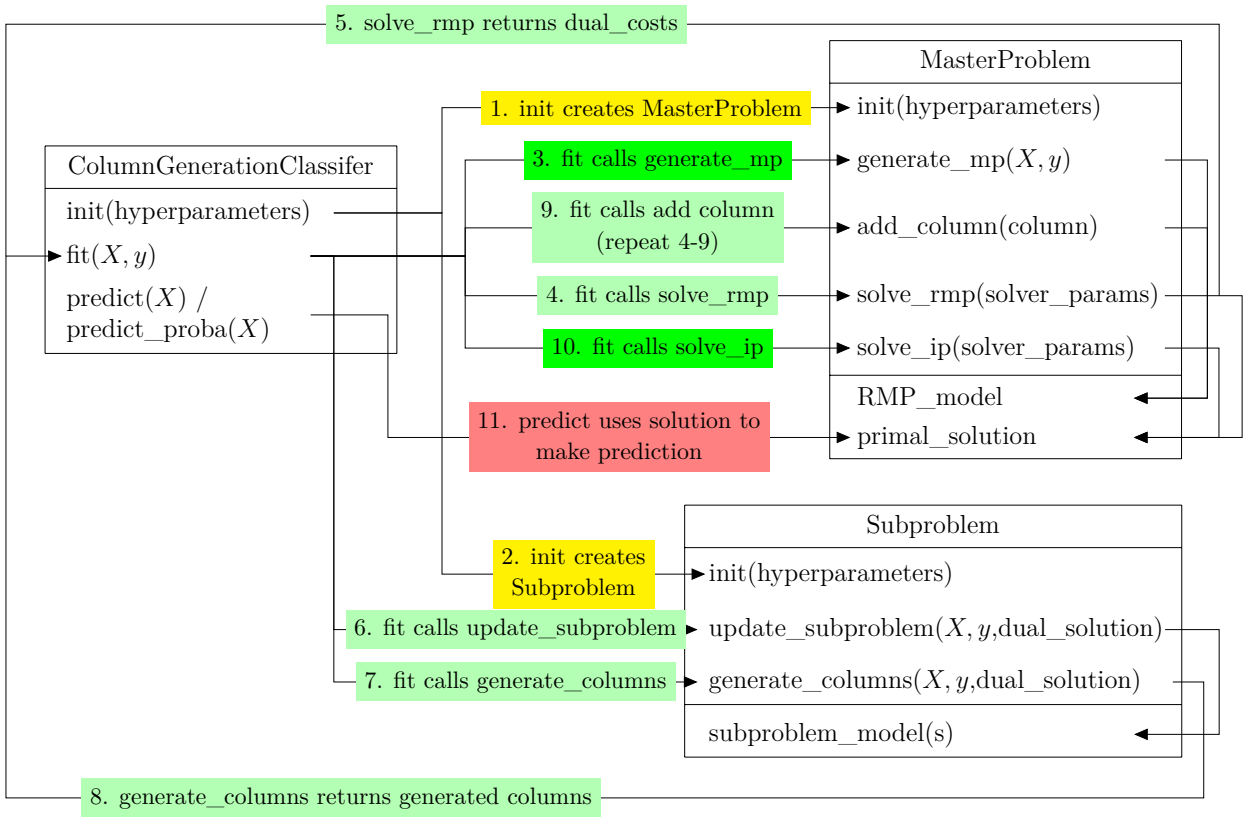

Figure 5.1 Interaction between different classes in the workflow of the CG Framework.

The classifier implements the *predict* method that extracts the generated boolean decision rule from the integer MP solution and checks whether the given data point satisfies the rule.

5.4.2 Decision Tree

As described in Chapter 4, we can use CG-based matheuristic to improve the performance of a decision tree classifier.

The MP uses the model (4.1). For this MP, a few hyperparameters control the cut generation: Whether to use the constraints (4.1c) as cuts, whether to generate new cuts, and the number of cut generation rounds in each iteration. The other hyperparameters include the problem-specific data: depth of the tree, set of split checks for each node, and set of targets. These are also the hyperparameters in the subproblems.

The heuristic path generation is implemented as a level 0 subproblem with empty *update_subproblem* method. The models (4.3) specific to each leaf are implemented as level 1 subproblems. These subproblems can be solved in parallel. The *update_subproblem* method ensures that each subproblem is updated in case the MP has generated new cuts.

The classifier implements the *predict* method that extracts the selected paths from the MP and checks which path the given data point follows. The predicted label is the target class of the path followed by the data point.

5.5 Exploiting Similarities in the Subproblems

The MP and the subproblem remain very similar across different CG iterations. Exploiting such similarities is relatively easy in the MP. In the MP, the only change includes added columns at each iteration. Using the primal simplex method, we can reuse the optimal basis from the last iteration for a warm start. The subproblems also remain similar across CG iterations. In the cases where the subproblems are MIPs (the examples described above), only the objective (expression of reduced cost for the solution) changes across iterations. The constraints and variable bounds remain unchanged.

This section discusses some techniques for reusing information generated by solving subproblems in previous iterations. Our approach was originally developed for the computational competition of the MIP workshop 2023 [71]. Our reoptimization approach (presented in Chapter 6) includes reusing primal solutions, reusing and improving branching information, automated parameter tuning, and turning off ineffective solver parts. Among these, the automated parameter tuning showed the biggest improvement in solving times. Hence, we implemented it in the framework as a starting point. The other techniques for reoptimization will be implemented in future releases of the package. The detailed algorithm for automated parameter tuning is provided in Section 6.5.

In the examples described above, we used Google OR-Tools CP-SAT solver with 8 threads to solve the subproblems. In this case, tuning the cut generation is ineffective, as the workers already use diverse parameters for the cuts. The tuned parameters for this solver are shown in Table 5.1.

Table 5.1 Tuned parameters for the CP-SAT solver.

Parameter	Effect	Tuned values	Default value
max_presolve_iterations	Limits the number of presolve iterations.	2,3,4	3
cp_model_probing_level	Controls the effort spent in probing.	1,2	2
min_num_lns_workers	Reserves threads for LNS workers.	0,2	2

The initial experimental results are promising. For the CG-based heuristic for training decision trees as described in Chapter 4, we measured the solving times of the subproblems that were called on at least 30 iterations. These experiments used the time limit of 10 minutes for the overall training process. We observed over 9% improvement in solving times for instances in the Default Credit, Magic, and Statlog satellite datasets. We also observed a degradation of solving time by up to 8% for instances in the HTRU 2 dataset. For longer experiments (time limit = 1 hour), the average solving times were improved by 4.8% for large datasets and by 3% for small datasets with the largest improvement of 19% in the Magic dataset and the worst degradation of 2% in the HTRU dataset. These results can be improved by tuning more parameters and tuning parameters for more values.

CHAPTER 6 ARTICLE 3: PROGRESSIVELY STRENGTHENING AND TUNING MIP SOLVERS FOR REOPTIMIZATION

Author: Krunal Kishor Patel

Submitted on January 24, 2024 in Mathematical Programming Computation.

Abstract

This paper explores reoptimization techniques for solving sequences of similar mixed integer programs (MIPs) more effectively. Traditionally, these MIPs are solved independently, without capitalizing on information from previously solved instances. Our approach focuses on primal bound improvements by reusing the solutions of the previously solved instances, as well as dual bound improvements by reusing the branching history and automating parameter tuning. We also describe ways to improve the solver performance by extending ideas from reliability branching to generate better pseudocosts. Our reoptimization approach, crafted for the MIP 2023 workshop computational competition, was honored with the first prize. In this paper, we thoroughly analyze the performance of each technique and their combined impact on the solver's performance. Finally, we present ways to extend our techniques in practice for further improvements.

Keywords: MIP, Reoptimization, Warmstart, Pseudocosts, Parameter-tuning

6.1 Introduction

Many applications require solving a sequence of similar mixed integer programs (MIPs). These instances differ on some specific parts (objective, right-hand side, variable bounds). Some of the major examples include decomposition algorithms (that use column generation or row generation) for which the subproblems are similar to each other except changes in the objective or right-hand side (see, e.g., [75], [76]). Some heuristics (e.g., [77], [78], and [79]) in MIP solvers also solve similar sub-MIPs. Furthermore, many applied problems need to be solved sequentially with changing data (see, e.g., [80], [81], [82], [83]). This paper proposes some techniques to reuse the information generated by solving previous instances to solve future instances more efficiently.

In this paper, we use the following formulation to denote a MIP:

$$\begin{aligned} min \quad & c^T x \\ s.t. \quad & Ax \leq b \\ & l \leq x \leq u \\ & x_j \in \mathbb{Z}, \forall j \in I \end{aligned} \tag{6.1}$$

with $A \in \mathbb{R}^{m \times n}, c \in \mathbb{R}^n, b \in \mathbb{R}^m, l \in (\mathbb{R} \cup \{-\infty\})^n, u \in (\mathbb{R} \cup \{\infty\})^n$, variables $x \in \mathbb{R}^n$ with $I \subseteq N = \{1, ..., n\}$.

The research on the topic of reoptimization of MIPs has been limited. Some MIP solvers (e.g., Gurobi [67], CPLEX [84]) use parts of primal solutions of the previously solved similar instances as warm start. The work of [85] and [86] describe reoptimization algorithms for certain classes of combinatorial optimization problems. In [87], the authors described a way to reuse the dual information. They mainly addressed the changes in the right-hand side. More recently, [88] investigated techniques for reoptimizaiton and implemented them in the SCIP solver [89], primarily focusing on changes in the objective function. A detailed description can be found in [76].

To solve the instances faster, we need to find good quality solutions faster (primal bound improvements) and prove the optimality faster (dual bound improvements). In this paper, we describe primal and dual bound improvement techniques by exploiting the similarity of the MIP instances. We focus on solving some series of 50 similar MIP instances that change in one or more of the following parts: objective coefficients, constraint bounds, variable bounds, and constraint matrix entries.

Our reoptimization approach, originally developed for the computational competition [74] of the MIP 2023 workshop, was honored with the first prize. The complete source code for this approach is available at
https://github.com/krooonal/mipcc2023/tree/MPCPaper.

Most of the techniques described in this paper are inspired by existing literature with some modifications. The major contribution of this paper is using these techniques together to achieve better performance. The techniques include

1. Reusing primal solutions: We describe how to combine solutions of the previous instances to form a partial solution hint that the MIP solver can convert into a feasible solution early in the search process. Some of these techniques are inspired by existing heuristics like RINS [77], Crossover [78], and CMSA [90].

2. Reusing branching information: MIP solvers progressively learn the effects of branching on each variable in the form of pseudocosts and other statistics, and we reuse these statistics as a warm start for the subsequent instances. This technique is also used in the context of restarts in MIPs [91].

3. Improving pseudocosts: Inspired by the main idea behind reliability branching [92], we explore how using more strong branching in the first instance affects the solution process of the subsequent instances.

4. Automated parameter tuning: Multi-armed bandit solution techniques are known for various hyperparameter tuning tasks in machine learning (see, e.g., [93]). In MIP solvers, multi-armed bandit algorithms are used in a node selection rule [94], and adaptive large neighborhood search (ALNS) [95](a primal heuristic for MIP solvers). We show how to use an algorithm for a multi-armed bandit problem (with slight modifications) to tune MIP solver parameters without any offline training.

5. Other optimizations: We detect the solver parts that are not working well for a given series of instances and turn them off. This technique can be seen as an extension of parameter tuning.

This paper focuses on reusing information generated by solving previous instances. We do not explore any offline learning approaches as enforced by the competition rules. Although, such offline learning methods may be used alongside our approach to get even better results.

This paper is organized as follows. Section 6.2 introduces the preliminary notations and the benchmarking system used in this paper. Section 6.3 describes techniques to improve the primal bound by reusing the solutions obtained for the previous instances. Sections 6.4 to 6.6 present techniques to improve the dual bound. Section 6.4 focuses on generating smaller branch-and-bound trees by reusing branching information. Section 6.5 describes a framework for automating the parameter tuning process. Section 6.6 focuses on exploring the branch-and-bound trees faster by turning off inefficient solver parts. Section 6.7 analyzes the performance of each technique and all of them together. Finally, Section 6.8 summarizes our approach and presents future research directions for reoptimization.

6.2 Benchmarking

Computational Resources

In this section, we discuss our computational experimental setup. We performed all the evaluations using SCIP optimization suite 8.0.2 [89] that includes SOPLEX 6.0.2 on a CPU

server with two sockets of Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz 28 cores each (total of 56 cores) and 512 GB of RAM. We used only one thread for each instance and a maximum of 16 GB RAM.

Dataset

The dataset described in this section will be used in all the experiments presented in this paper. These instances were used in the MIP Computational Competition 2023. There are 15 series. Each series has 50 similar instances with one or more components changing across instances. For each series, we solve the instances in the provided sequence, reusing only the information generated by solving previous instances in the series.

The names of the series suggest the changing components. For each series, we used a time limit for solving each instance in that series. Table 6.1 describes the changing components, time limit (in seconds), and shifted geometric means (shifted by 10 seconds) of solving times without using any reoptimization techniques for each series.[1]

Table 6.1 Changing components and time limits of different series.

Series	Changing component(s)	Time limit per instance	Solving times
Bound 1	l, u	600	400.48
Bound 2	l, u	300	186.23
Bound 3	l, u	600	336.96
RHS 1	b	400	294.39
RHS 2	b	60	69.55
RHS 3	b	550	307.66
RHS 4	b	60	70.00
Objective 1	c	400	301.95
Objective 2	c	300	256.69
Objective 3	c	350	153.85
RHS Objective 1	b, c	500	492.80
RHS Objective 2	b, c	250	91.03
Matrix 1	A	300	270.70
Matrix RHS Objective Bound	A, b, c, l, u	180	92.80
Matrix RHS Bound	A, b, l, u	400	204.31

The detailed description of the origin of each series is provided in [74]. The series Bound 1, Bound 2, Bound 3, Objective 2, and RHS 3 are generated by mutating some instances

[1]For the series RHS 2 and RHS 4, the shifted geometric means of solving times are higher than the time limit. This indicates that SCIP could not solve most of these instances in the given time limit.

from the MIPLIB 2017 benchmark library [96]. Other series are based on various theoretical approaches (including stochastic multiple binary knapsack problem [97], column-generation-based heuristic [1], and constraint generation for MIP [98]) and real-world applications (including hydro unit commitment problem [80], multilevel supply chain [81], a hydroelectric valley industrial use case [82], and optimal vaccine allocation problem [83]).

Scoring

For each instance in a series, we compute a 'Total score' to measure the solver's performance on that instance, as used in the MIP Computational Competition 2023. The 'Total score' is the sum of two components with equal weights: 'Time score' and 'Gap score'.

The 'Time score' is the fraction $\frac{\text{time spent}}{\text{time limit}}$ if the instance is solved to optimality and 1 otherwise. The 'Gap score' is defined as the fraction $\frac{|pb - db|}{\max(|pb|, |db|)}$, where pb and db are the primal and dual bounds, respectively. The 'Gap score' is 1 if the primal or dual bound are infinite or have different signs, i.e., $pb \cdot db < 0$. When the instance is solved to optimality, the 'Gap score' is 0, and when the 'Gap score' is positive, the 'Time score' is 1 as the solver failed to prove optimality in the given time limit.

The competition also included a 'Nofeas score' in the total score, which is equal to 1 if the solver failed to find any feasible solution for the instance (implying 'Time score' and 'Gap score' to be 1). In all our experiments, the solver always found at least one feasible solution for each instance, so this component was 0 for all instances; hence, we removed it for simplicity.

For each experiment, we report the average instance total score for batches of 10 instances (1-10, 11-20, 21-30, 31-40, 41-50). The average total score for each batch helps us understand the effect of the reoptimization techniques as we progress through the series.

6.3 Reusing Primal Information

When solving multiple similar instances, we expect the solutions of those instances to be similar as well. In this section, we describe various ways to use the solutions of the instances solved before.

Modern MIP solvers aim to derive a feasible solution from any complete or partial solution provided to them (denoted as a 'solution hint' or 'hint' henceforth) at the beginning of the search process. In SCIP, this is done by a heuristic called *completesol*. SCIP can take multiple solution hints at the beginning of the search process and the *completesol* heuristic processes each hint sequentially. The heuristic checks for feasibility if the hint represents a complete solution. If feasible, the solution is added to the solution pool when it improves

the incumbent. For a partial solution, the heuristic attempts to complete it by solving a sub-MIP. To our knowledge, the heuristic does not attempt to repair an infeasible solution hint.

Unless only the objective function has changed (i.e., all feasible solutions of the previous instances are also feasible for the current instance), we update the node limit (number of nodes it can use in the sub-MIP) for this heuristic to 5000 (default limit = 500) and also remove the limit on the maximum number of improving solutions (default limit = 5). These changes ensure that the *completesol* spends more effort to convert a given solution hint into a feasible solution.

Removing the limit on the maximum number of improving solutions allows us to provide as many solution hints as possible. However, we expect the best-found solutions of the previous 10 instances to represent well the solution pattern (or trend) of the series. Hence, we decided not to use more than 10 solution hints in any series (5 for the Objective series) for our evaluations. If the solution hints help improve the total score, one can experiment with providing more solution hints.

First, we evaluated with providing the complete best-found solutions for the previous 10 instances (if available). For series where only the objective has changed, we provide the complete solutions for the previous 5 instances to reduce the efforts spent by the *completesol* heuristic.

Next, we remove the continuous variables' values from the hint (making the solution a partial solution) to improve the number of hints getting converted into a feasible solution (hint-to-solution conversion number).

Furthermore, when the variable bounds change in a series, we clip the solution values by the bounds of the current instance. This means that, if a variable's value in the hint is greater than its new upper bound, we set the value to the new upper bound in the hint. We use similar clipping for the lower bound.

Finally, we construct a common partial solution to improve the hint-to-solution conversion number as follows. We record the best-found solutions of the instances. We add a variable-value pair to the hint that is part of the first instance solution and present in at least $\alpha\%$ of the (best-found) solutions of all previous instances. The variable-value pairs being part of the first instance solution ensures that the provided hint is feasible for at least one instance in the series and reduces the chances of combining inconsistent variable-value pairs for the entire series. This idea of combining multiple solutions is inspired by many existing heuristics like RINS [77], Crossover [78], and CMSA [90].

The α value controls the amount of effort needed to find a feasible solution (using the given solution hint) and the solution's quality. After preliminary testing of different values of $\alpha \in \{80, 90, 95, 100\}$, we selected the value $\alpha = 90$ for our evaluations. We did not experiment with dynamically changing α for a series.

For our final evaluation, we provide a common partial solution hint and the clipped integer solutions of the previous 9 instances (4 instances for the Objective series).

To measure the effect of each technique, we record the percentage of instances for which the provided hints are converted into at least one feasible solution by the *completesol* heuristic. The conversion percentages are reported in Table 6.2.

Complete solution hints are converted into feasible solutions for all instances in all Objective series. The conversion percentages are above 80% for both RHS Objective series. For the other series, the conversion numbers are very low, as expected.

Removing continuous variable values from the hint along with bound clipping slightly improves the conversion percentages. The bound clipping only affects five series where the variable bounds are changing. The improvement in the conversion percentage is largely due to the removal of continuous variables' values.

Providing a common solution hint significantly improves the conversion percentages. However, the conversion percentages remain 0 for RHS series 3 and Matrix RHS Bound series.

Finally, we show the effect of providing solution hints on the total scores for each series in Table 6.3. The conversion rate column shows the percentage of instances for which the solution hints were converted into at least one feasible solution by the solver (same as the last column in Table 6.2). The BASE column shows the total scores of the variant where we do not provide any solution hints, and the SOLHINT column shows the percentage improvement over BASE scores for the variant where we provide the Common Clipped Integer solution hints as described above. The detailed batch-wise results are presented in Table A.1.

In the series Bound 3, RHS 1, RHS 2, RHS 4, Objective 1, RHS Objective 1, and Matrix RHS Objective Bound, the conversion rates are above 50%, and we observe a gain in the total scores. For the other series, it is unclear whether providing solution hints is helpful. Specifically for series Bound 1 and Matrix 1, providing solution hints significantly degrades the total scores even though the conversion rate is high. We address this issue in Section 6.5.

Table 6.2 Percentages of instances for which the provided hints are converted into at least one feasible solution by *completesol* heuristic for different types of hints. The highest numbers are in bold.

Series	Complete solution (%)	Clipped Integer solution (%)	Common Clipped Integer solution (%)
Bound 1	40.8	40.8	**51.0**
Bound 2	4.1	6.1	**42.9**
Bound 3	8.2	10.2	**61.2**
RHS 1	0.0	0.0	**69.4**
RHS 2	**100.0**	**100.0**	**100.0**
RHS 3	0.0	0.0	0.0
RHS 4	**100.0**	**100.0**	**100.0**
Objective 1	**100.0**	**100.0**	**100.0**
Objective 2	**100.0**	**100.0**	**100.0**
Objective 3	**100.0**	**100.0**	**100.0**
RHS Objective 1	81.6	85.7	**93.9**
RHS Objective 2	83.7	**89.8**	**89.8**
Matrix 1	0.0	**100.0**	**100.0**
Matrix RHS Objective Bound	65.3	65.3	**87.8**
Matrix RHS Bound	0.0	0.0	0.0
Average	52.2	59.9	**73.1**

6.4 Reusing and Improving Branching Information

6.4.1 Reusing Branching Information

MIP solvers spend a significant amount of time proving the solution's optimality. The MIP solvers record various branching-related statistics in the branch and bound process that are used in branching heuristics.

SCIP stores the branching statistics for each variable in the variable history data structure. This structure stores many branching-related records, including the variable's pseudocost information, conflict records, and inference records.

For each instance in a series, we retrieve each variable's history. We pass this variable history to the corresponding variable in the next instance at the beginning of the solving process. Since the instances are similar, we expect this information to also be useful for the next instance, and the solver no longer needs to compute the branching statistics from scratch.

Table 6.3 Effects of providing solution hints on total scores. Average BASE scores and % improvements caused by SOLHINT variant. Positive improvement numbers are in bold.

Series	Conversion rate (%)	BASE	SOLHINT (% improvement over BASE)
Bound 1	51.0	0.734	-10.90
Bound 2	42.9	0.658	-0.30
Bound 3	61.2	0.636	**2.52**
RHS 1	69.4	0.746	**2.41**
RHS 2	100.0	0.994	**2.82**
RHS 3	0.0	0.652	-0.31
RHS 4	100.0	1.000	**1.60**
Objective 1	100.0	0.756	**4.50**
Objective 2	100.0	0.910	-3.52
Objective 3	100.0	0.420	-1.43
RHS Objective 1	93.9	0.996	**4.42**
RHS Objective 2	89.8	0.418	**0.48**
Matrix 1	100.0	0.896	-6.47
Matrix RHS Objective Bound	87.8	0.504	**28.17**
Matrix RHS Bound	0.0	0.572	0.00
Average	73.1	0.726	**1.60**

Along with branching statistics for each variable, SCIP maintains a 'global history' structure containing the same statistics for the entire problem (i.e., statistics averaged over all variables). When the number of branching records for a particular variable is very low, SCIP uses the records stored in global history for that variable while making branching decisions. We also extract this global history structure and pass it to the next instance.

The current state-of-the-art branching rules heavily rely on the pseudocosts for variables [92]. Pseudocosts measure the average objective improvement for a unit change in value for every integer variable. Their use in branching was first reported in [99]. For each variable, the solver maintains the pseudocost and updates it incrementally by storing the number of updates as the pseudocost count in the history structure.

One of the most commonly used branching rules is reliable pseudocosts [92]. At the beginning of the solving process, the pseudocosts (and other records in the variable history) are unreliable since the number of pseudocost updates is low [100]. To address this, the reliable pseudocosts branching rule uses strong branching [101] at the beginning of the solving process. In strong branching, the solver evaluates multiple variables for branching at each

node and updates their pseudocosts. Hence, even though the branching rule takes longer to execute, the variables receive more pseudocost updates. This is required to have reliable pseudocosts.

When we pass the history to the next instance, we reduce the pseudocost count to 4 if it is greater than 4. The idea behind this reduction is to make the solver use the pseudocosts of the previous instance as a 'warm start' only. This allows the pseudocosts to converge to the actual average value (which can be different for the next instance) faster. This is inspired by [91]. We used the value 4 for reducing the count as a threshold because it is slightly less than the fixed number threshold 5, after which pseudocosts are considered 'reliable' in SCIP [102]. We expect such a warm start to only reduce the computing time by reducing the number of strong branching calls and not the number of branching nodes explored.

6.4.2 Improving Pseudocosts

We extend the reliability branching idea further and solve the first instance in the series with full strong branching to generate better pseudocosts. Since we reuse the pseudocosts across different instances, generating more reliable pseudocosts is essential. Solving the instance with strong branching helps us generate more reliable pseudocosts.

Furthermore, we observed that the pseudocosts are more reliable when the objective and variable bounds are unchanged. In that case, we switch to the pseudocost branching rule instead of the default reliable pseudocost branching rule after solving the first instance with strong branching.

Using full strong branching comes at a cost. The solving process is significantly slower for the first instance, where we use full strong branching. In only three series (compared to nine in BASE approach), we could solve the first instance to optimality using strong branching: RHS Objective 2, Matrix 1, and Matrix RHS Bound. The average gap score for the first instance across all series is 0.18 when we use strong branching and 0.003 when we do not.

However, we expect the recorded pseudocosts to be more reliable and compensate for the loss on the first instance by solving subsequent instances faster. In the competition, the total score for the first instance in a given series was given smaller weights compared to the subsequent instances, motivating us to experiment with more exploration for the initial few instances.

Table 6.4 shows the effect of reusing branching information and using strong branching for the first instance on the total scores. The BASE column shows the total scores of the variant where we do not reuse any branching information, the REHIS column shows the percentage

improvement over BASE total scores of the variant where we reuse the variable and global history, and the column REHIS+SB shows the percentage improvement over BASE total scores for the variant where we reuse the variable and global history and solve the first instance with strong branching. The detailed batch-wise results are presented in Table A.2.

The scores in Table 6.4 suggest that reusing branching history improves the solver's performance in general. However, it is not better when the objective is changing. This is expected because we should not expect the pseudocosts to remain similar across instances where the objective changes. Using strong branching for the first instance is better on all RHS series. We do not have a good explanation for why using strong branching is most effective on series with RHS changing. For the Bound and the Matrix series, there is not a clear improvement in the scores over the REHIS variant.

Table 6.4 Effects of reusing branching history on total scores. Average BASE scores and % improvements caused by REHIS and REHIS+SB variants. Better positive improvement numbers are in bold.

Series	BASE	REHIS (% improvement over BASE)	REHIS+SB(% improvement over BASE)
Bound 1	0.734	**34.06**	30.25
Bound 2	0.658	**12.77**	6.08
Bound 3	0.636	**4.72**	3.14
RHS 1	0.746	6.43	**7.77**
RHS 2	0.994	0.00	**6.64**
RHS 3	0.652	7.98	**14.42**
RHS 4	1	0.00	**5.80**
Objective 1	0.756	2.38	**3.17**
Objective 2	0.91	-6.15	-11.21
Objective 3	0.42	-12.38	-15.71
RHS Objective 1	0.996	**0.20**	-1.61
RHS Objective 2	0.418	-9.57	-3.35
Matrix 1	0.896	-1.79	-4.25
Matrix RHS Objective Bound	0.504	-19.44	-17.06
Matrix RHS Bound	0.572	-0.69	**1.39**
Average	0.726	1.23	**1.70**

6.5 Automated Parameter Tuning

MIP solvers contain many tunable parameters, and since we are solving similar instances in a series, we want to automate the parameter tuning process to improve the solving speed

for the later instances. The parameter tuning with offline training has been explored before (see, e.g., [103]). This section focuses on MIP solver parameter tuning without any offline training.

For a given parameter, we consider two values for tuning (i.e., ON and OFF) and maintain scores for those values. The score computation is inspired by the Upper Confidence Bound (UCB) Algorithm [104] from reinforcement learning. We compute the score S_v for a value v using equation (6.2).

$$S_v = Q_v + C/N_v \tag{6.2}$$

This score involves two parts: (a) The running average of the base score (Q_v), and (b) The uncertainty of the score, i.e., the confidence bound ($1/N_v$), where N_v is the number of score updates the value has received. When a value is not explored enough, the confidence bound adds a higher number to the score to encourage more exploration for that value. C is a weight constant that determines how fast the score converges. Based on our experiments, the value $C = 0.3$ gives the best results regarding convergence correctness (i.e., the desired value being used more) and convergence speed. For the base score, we use the negative total score of the instance, as described in Section 6.2.

In the UCB algorithm, the confidence bound uses the square root of N_v and the total number of score updates across all values. Using the total number of score updates ensures that confidence in the score of a value decreases over time. These parts are helpful in general. However, we discard the total number of score updates for the competition and use N_v instead of its square root for faster convergence. See Table 6.5 for a comparison of convergence speed. Using $\sqrt{N_v}$ results in more exploration even in the last batch (instances 41-50) for multiple values of C, which is not desired for the competition but may be useful in practice when the total number of instances is significantly more than 50.

Table 6.5 Convergence comparison between C/N_v and $C/\sqrt{N_v}$.

	C=0.2		C=0.3		C=0.4	
N_v	C/N_v	$C/\sqrt{N_v}$	C/N_v	$C/\sqrt{N_v}$	C/N_v	$C/\sqrt{N_v}$
1	0.200	0.200	0.300	0.300	0.400	0.400
5	0.040	0.089	0.060	0.134	0.080	0.179
10	0.020	0.063	0.030	0.095	0.040	0.126
20	0.010	0.045	0.015	0.067	0.020	0.089
30	0.007	0.037	0.010	0.055	0.013	0.073
40	0.005	0.032	0.008	0.047	0.010	0.063
50	0.004	0.028	0.006	0.042	0.008	0.057
100	0.002	0.020	0.003	0.030	0.004	0.040
200	0.001	0.014	0.002	0.021	0.002	0.028
500	0.000	0.009	0.001	0.013	0.001	0.018

The tuning involves exploration and exploitation stages. The score value S_v already considers the number of score updates and motivates some exploration. In addition, we mark a value under exploration if it has not been used at least four times. For a given parameter, we prioritize selecting the values under exploration, if any.

For the exploitation stage, selecting the value with the highest score S_v is sufficient. However, the UCB algorithm assumes that the underlying reward distribution is constant. This would be the case if all instances have similar difficulties. However, this is not always true. The instance difficulties may change in a series. To address that, we also use the standard deviation of the base score. For the selection, we mark any values v with a score S_v within a distance 1/10 of the standard deviation from the highest score S_v^* as candidate values. If there are multiple candidate values, we select the final value uniformly at random among them.

In this framework, one can tune multiple parameters at the same time. However, doing so increases the noise, and the scores converge slowly. We tune three binary parameters in our approach.

1. Provide hint or not
2. Use root node cuts or not
3. Use cuts at other nodes or not

In general, providing hints helps the solver. However, as shown in Section 6.3, in some cases, providing solution hints causes harm in the solving process. Hence, we decided to use our parameter tuning framework to decide whether to provide solution hints to the solver. For this parameter, we update the score for value 'ON' (provide hint) only if the solver converts the hints into at least one feasible solution. Otherwise, we update the value 'OFF' score if we do not provide hints or the solver fails to convert the hints into a feasible primal solution.

Similarly, in most cases, using cuts is helpful. However, adding cuts can hurt the performance in some instances. We expect the effects of adding cuts to remain similar throughout the entire series, i.e., it either improves or degrades performance for all instances. But, we expect the effect to differ between root node cuts and other node cuts. The root node cuts are always global (i.e., valid for the entire tree), and solvers typically spend more effort generating cuts at the root node. For the cuts parameters above, we do not check if the solver adds the cuts when the 'ON' value is selected, unlike the solution hint parameter.

When tuning multiple parameters, we perform initial exploration in a deterministic fashion. The first parameter selects the value 'ON' on every alternate instance during exploration. The second parameter selects the value 'ON' on instances with the second bit set on the binary representation of the instance index. The third parameter selects the value 'ON' on instances with the third bit set on the binary representation of the instance index. Table 6.6 shows how the values are selected for each parameter during deterministic exploration if tuning of all parameters starts at instance 0.

Table 6.6 Selected parameter values during deterministic exploration.

Instance index	Solution hint	Cuts	Root cuts
0	OFF	OFF	OFF
1	ON	OFF	OFF
2	OFF	ON	OFF
3	ON	ON	OFF
4	OFF	OFF	ON
5	ON	OFF	ON
6	OFF	ON	ON
7	ON	ON	ON

For the computational experiments, we measure the effects of parameter tuning in the presence of other reoptimization techniques discussed in Sections 6.3 and 6.4. The parameters'

tuned values may differ if the other reoptimization techniques are not used. We noticed that disabling restarts gives better performance in most cases. The probable cause might be that the solver is already reusing much information from the previous solutions. Hence, for the computational experiments, we turn off restarts.

Table 6.7 shows the most selected value for each parameter for each series. The count column shows the number of times that value was used out of 49. We do not count the parameter values used for the first instance. The count values suggest how quickly the above system converged for the selected values. Some parameters do not significantly affect the total scores and did not converge. For example, the selected values for the solution hint parameter in Bound 2, Objective 2, and Objective 3 series have counts close to 25. The count values above 40 in most cases suggest a strong preference, and count values between 30 and 40 suggest a weak preference for a parameter value. Note that these preferences may not always be accurate (i.e., we get better scores when we use the non-preferred values.)

Because of the initial deterministic exploration, we expect the max count value to be 45. However, the count values for the solution hint parameter are 49 for the RHS 3 and Matrix RHS Bound series. This happens because the count for the 'ON' value in this parameter does not represent how many times the hints were provided but how many times the hints were converted into a feasible solution. None of the solution hints were converted into a feasible solution for the RHS 3 and Matrix RHS Bound series (see Table 6.2). Hence, even when the hints were provided, the scores of the 'OFF' values were updated. Since the counts for the 'ON' value never reached count 4, it remains marked under exploration, and the tuning algorithm keeps using it.

Table 6.7 Selected values and count for the tuned parameters for each series.

Series	Solution hint		Cuts		Root cuts	
	Value	Count	Value	Count	Value	Count
Bound 1	OFF	45	ON	45	ON	45
Bound 2	OFF	27	ON	45	ON	45
Bound 3	OFF	45	ON	45	ON	45
RHS 1	ON	33	ON	45	OFF	41
RHS 2	ON	45	OFF	41	OFF	41
RHS 3	OFF	49	ON	40	OFF	34
RHS 4	ON	45	OFF	41	OFF	41
Objective 1	OFF	28	OFF	32	OFF	29
Objective 2	ON	27	ON	43	ON	37
Objective 3	ON	39	ON	45	OFF	41
RHS Objective 1	ON	30	OFF	41	OFF	41
RHS Objective 2	ON	36	OFF	28	ON	45
Matrix 1	ON	38	ON	45	ON	44
Matrix RHS Objective Bound	ON	40	ON	45	OFF	41
Matrix RHS Bound	OFF	49	ON	45	ON	45

Table 6.8 shows the effect of automated parameter tuning on the total scores. The column NOTUNING shows the total scores of the variant where we do not tune the parameters, and the column TUNING shows the total scores of the variant where we automate the parameter tuning as described above. For both variants, we provide the solution hints as described in Section 6.3 and reuse the branching information as described in Section 6.4. For both variants, we turn off restarts in the solver. The detailed batch-wise results are presented in Table A.3.

In most series, the variant TUNING performs worse in the first batch. This is because of the exploration stage of the tuning process. The TUNING variant scores for the later batches are better compared to the NOTUNING variant in general. For the series Bound 3, RHS Objective 2, and Matrix 1, the TUNING variant has worse scores than the NOTUNING variant because the TUNING variant converged to incorrect values for some parameters (see the batch-wise comparison in Table A.3). In the Objective 2 and the RHS 3 series, it is unclear if the TUNING variant converged to the correct values. In the series Bound 2 and Matrix RHS Bound, the TUNING variant improves the total score in the subsequent batches, but

it does not cover the losses made in the exploration stage. In the other series, the TUNING variant clearly outperforms the NOTUNING variant.

Table 6.8 Effects of automated parameter tuning on total scores. Average NOTUNING scores and % improvements caused by TUNING variant. Positive improvement numbers are in bold.

Series	NOTUNING	TUNING (% improvement over NO-TUNING)
Bound 1	0.7	**26.86**
Bound 2	0.444	-3.60
Bound 3	0.484	-14.05
RHS 1	0.672	**3.87**
RHS 2	0.664	**12.35**
RHS 3	0.596	-5.03
RHS 4	0.678	**13.57**
Objective 1	0.744	**11.02**
Objective 2	1.008	-2.18
Objective 3	0.656	**38.72**
RHS Objective 1	0.992	**6.05**
RHS Objective 2	0.572	-27.62
Matrix 1	1.016	-2.56
Matrix RHS Objective Bound	0.466	**14.16**
Matrix RHS Bound	0.726	-6.61
Average	0.695	**4.33**

6.6 Other Optimizations

MIP solvers have many components turned on by default. Not all of them perform well during the solving process. After solving each instance, we extract the statistics for primal heuristics, presolvers, and separators (cut generators). If they are unsuccessful for the initial few instances and expensive in solving time, we turn them off to give the other components some extra time. Turning such non-performing components off had a minor improvement in the total scores.

We use the following criteria to mark a component unsuccessful.

- A presolve rule is unsuccessful if the total number of changes made in the first 15 instances by that rule is 0.

- A separator is unsuccessful if the total number of generated cuts in the first 25 instances by that separator is 0.
- A primal heuristic is unsuccessful if it either fails to find any solution in the first 25 instances or the time spent per best solution found is more than 20% of the given time limit for a single instance (measured after the first 25 instances).

We show the effect of turning inefficient solver parts off on total scores in Table 6.9. For this experiment, the other reoptimization techniques discussed in Sections 6.3, 6.4, and 6.5 were also in use, ensuring that we only turn off a solver part if it is ineffective under various settings (different parameter values and improved pseudo costs). The detailed batch-wise results are presented in Table A.4.

Unlike previous experiments, turning the ineffective components does not consistently improve the total scores for most series. However, for series RHS 2, RHS 4, Objective 1, and RHS Objective 2, we see a significant improvement in the total scores of later batches. On the other hand, the total scores for RHS 3, Objective 3, and Bound 3 show significant degradation in most batches, suggesting that we turned off one or more essential components of the solver that were ineffective in the earlier instances of that series. For the other series, it is unclear whether the detection of ineffective parts was successful (see the batch-wise comparison in Table A.4).

6.7 All Together

Table 6.10 shows the average total scores of the different techniques we used for reoptimization. The first row shows the average total score of solving each instance from scratch (Base). Reusing the primal solutions of previous instances has the least effect (1.30%) of all the techniques described in this paper. Reusing branching history along with using strong branching for the first instance improves the performance slightly more (2.57%). The most considerable improvement (of 4.53%) comes from the parameter tuning, which is more than using the primal solution hints and reusing branching history combined (4.40%). Finally, combining all techniques results in a 12.77% improvement over the Base approach.

Furthermore, we show the performance of the reoptimization techniques per batch in Table 6.11. Compared to the Base approach, our approach gives worse scores in the first batch by almost 4%. This slowdown is caused mainly by the parameter tuning exploration stage (see Table A.3). Using strong branching for the first instance also contributes to this slowdown but not as much as parameter tuning (see Table A.2). However, our approach progressively

Table 6.9 Effects of automated parameter tuning on total scores. Average NOTURNOFF scores and % improvements caused by TURNOFF variant. Positive improvement numbers are in bold.

Series	NOTURNOFF	TURNOFF (% improvement over NO-TURNOFF)
Bound 1	0.508	**5.51**
Bound 2	0.434	-5.07
Bound 3	0.52	-2.69
RHS 1	0.66	**3.64**
RHS 2	0.57	**24.56**
RHS 3	0.584	-16.10
RHS 4	0.582	**21.65**
Objective 1	0.648	**2.78**
Objective 2	1.046	**1.72**
Objective 3	0.398	-8.04
RHS Objective 1	0.898	-3.79
RHS Objective 2	0.686	**5.54**
Matrix 1	1.034	-1.35
Matrix RHS Objective Bound	0.36	-2.22
Matrix RHS Bound	0.766	**0.52**
Average	0.646	**1.78**

strengthens and tunes the solver in the later batches. In the last three batches, the scores improve by more than 15% compared to the Base approach. In the last batch, we observe an improvement of 22% in the total scores compared to the Base approach.

We expect this improvement trend to slow down if we have more instances. We designed the approach to complete most solver tuning and improvements by the beginning of the last batch (41-50) and exploit what works well after that. However, this approach can be modified for real-world settings with more than 50 instances in a series. We talk more about such modifications in Section 6.8.

Finally, we show the shifted geometric means of solving time (shifted by 10 seconds) for instances in each series in Table 6.12. Compared to the Base approach, our approach significantly improves the solving times for all series except RHS Objective 2, Matrix 1, and Matrix RHS Bound. In RHS Objective 2 and Matrix RHS Bound series, the biggest degradation is caused by automated parameter tuning. In Matrix 1, all of the techniques presented above degraded the performance. The biggest degradation is caused by reusing the primal

solutions, which the automated parameter tuning failed to stop. This suggests the need to improve the parameter tuning approach further.

Table 6.10 Average total scores of different techniques used for reoptimization.

Approach	Average total score	Improvement (%)
Base	0.727	0.00
Primal solution hint	0.717	1.30
Reusing branching history	0.712	2.09
Reusing branching history with strong branching	0.708	2.57
Primal solution hint + Reusing branching history with strong branching	0.695	4.40
(All above +) parameter tuning	0.662	8.93
(All above +) Other optimizations	0.634	12.77

Table 6.11 Average total scores of reoptimization techniques per batch.

Batch	Average total score		Improvement (%)
	Base	Reoptimization	
1-10	0.688	0.715	-3.91
11-20	0.737	0.672	8.74
21-30	0.718	0.601	16.29
31-40	0.732	0.593	18.88
41-50	0.759	0.588	22.57

6.8 Conclusion and Future Work

In this paper, we described some techniques to strengthen and tune the MIP solver by reusing information generated by solving previous instances. The datasets we used to measure the impact contained 50 similar instances for each series, as employed in the MIP computational competition 2023. This benchmark's intentional limitation to relatively small datasets, solvable within 10 minutes using commercial solvers, facilitated efficient experimentation and evaluation of various reoptimization techniques. However, in practical scenarios, the number of instances within a series can be significantly larger, posing a challenge for traditional methods. Below, we summarize each technique and propose strategies to overcome these limitations.

Table 6.12 Shifted geometric means of solving time for instances for each series.

Series	Solve time geometric means		Improvement (%)
	Base	Reoptimization	
Bound 1	400.48	215.99	46.07
Bound 2	186.23	117.68	36.81
Bound 3	336.96	239.61	28.89
RHS 1	294.39	215.97	26.64
RHS 2	69.55	33.94	51.20
RHS 3	307.66	233.51	24.10
RHS 4	70	34.46	50.77
Objective 1	301.95	232.44	23.02
Objective 2	256.69	146.41	42.96
Objective 3	153.85	97.77	36.45
RHS Objective 1	492.8	319.52	35.16
RHS Objective 2	91.03	109	-19.74
Matrix 1	270.7	298.94	-10.43
Matrix RHS Objective Bound	92.8	56.99	38.59
Matrix RHS Bound	204.31	236.29	-15.65

In this paper, we tested our approach on a relatively small benchmark. In practice, the number of instances can be much larger than 50 for a given series. Moreover, all the instances in our benchmark can be solved within 10 minutes by any commercial solver, hence, they are relatively easy to solve. The competition committee introduced these limitations intentionally to limit the time required to experiment and evaluate various reoptimization techniques. Below, we suggest some ways to extend our approach to address these limitations.

First, we described how to reuse parts of previous instance solutions. Constructing a common partial solution and providing it as a solution hint works best in terms of converting the hints to a feasible solution. We use variable-value pairs that appear in at least $\alpha\%$ of all the solutions to generate the common solution. One potential improvement to our approach is to dynamically change the value of α based on the hint-to-solution conversion numbers. This can potentially improve the conversion numbers.

As discussed in Section 6.7, most of the performance improvement comes from reusing the information to improve the dual bounds faster. Reusing the branching history alone brings more significant improvements than reusing the primal solutions. Pseudocosts can also be reused in the form of branching priorities for variables provided to the solver along with the model. We did not explore this direction.

We use strong branching to solve the first instance to generate more reliable pseudocosts

for the subsequent instances. This technique resulted in significant gains in the RHS series. However, the effects of using strong branching in the first instance faded for the instances in later batches. In many series, using strong branching prevented us from proving optimality in the first instance. An alternative approach can be to consider using reliability branching with increased thresholds for marking the pseudocost of a variable reliable. If the time limits are not strict, we can also consider using more time for the first few instances to accommodate more strong branching than the default parameters. The idea is to perform more strong branching in the initial instances to generate more reliable pseudocosts. Furthermore, we only use strong branching for the first instance. One can consider using more strong branching after solving every n instances.

Automated parameter tuning is the most extensible technique that also leads to the most significant improvements in performance. Since we have only 50 instances in each series, our tuning approach focuses on faster convergence. We only tune 3 parameters as tuning more parameters simultaneously is not effective with our approach. However, if we have more instances, we can consider tuning more parameters sequentially, and more exploration (leading to slower but more reliable convergence) for each parameter tuned.

Finally, we collect statistics of each major solver component and automatically disable some ineffective ones. For some series, this significantly improves the performance. However, the improvements are not consistent.

It is important to note that not every technique we discussed in this paper works well on each series. While each technique improves the average scores, they may significantly degrade the performance on some series. Such mixed results can be partly explained by the complexity of the MIP solvers. MIP solvers have many components, and the interplay between them makes it hard to predict whether a particular technique will work well.

Except for reusing the branching history, the techniques discussed in this paper do not require access to the solver code as most solvers have public APIs for providing hints, statistics collections, and on/off control of various parts through parameters. We did not use any domain-specific knowledge in our approach apart from the limited use of the knowledge of the changing parts in each series. This means that the techniques we presented can be used for any reoptimization setting. Furthermore, we can get better results by using domain-specific knowledge along with these techniques.

CHAPTER 7 CONCLUSION

In this chapter, we present the summary of our work in Section 7.1 and future research directions in Section 7.2.

7.1 Summary of Works

This thesis focused on exploring the use of CG-based classifiers within the domain of ML and addressing practical challenges associated with applying them in practice.

In Chapter 3, we expanded the CG-based approach for creating boolean decision rules to a practical scenario within aviation that entails multiclass classification. We extended the methods to address challenges associated with one-vs-rest classification, including dealing with multiple outputs and class imbalances. We introduced a heuristic utilizing a CP-SAT solver to expedite CG and outlined a method for reducing the search space in the subproblem. The accuracy we achieved using this method were comparable to some of the ML algorithms like SVM and small neural networks. While the CG-based approach cannot match the accuracies achieved by advanced ML algorithms like BERT, the loss in accuracy is strongly balanced by the explanation.

In Chapter 4, we presented modifications to the CG-based heuristic of [1] for training classification trees. First, we changed the subproblem model, significantly decreasing the number of subproblems in multiclass classification scenarios. We showed that this change resulted in faster generation of columns. Following that, we demonstrated that the data-dependent constraints within the MP are implied and utilized them as cutting planes. We showed that this change significantly reduces the training times without loss in accuracy. Additionally, we outlined a separation model designed to generate data points where the LP relaxation solution violates their corresponding constraints. This allowed us to prove optimality of the MP at the root node without using branch-and-price for most instances. Finally, we presented a preprocessing and initialization routine to reduce the size of both master and subproblems, consequently cutting down on training times for large datasets. Overall, we demonstrated that these changes result in higher accuracy gain over CART compared to the original approach in [1] on both training and testing datasets.

In Chapter 5, we detailed a software framework designed to simplify the process of developers in creating and implementing CG-based ML models on top of scikit-learn. The framework encompasses shared components utilized in different CG-based classifiers. Additionally, we

incorporated methods to leverage similarities within subproblems, enhancing the speed of solving these subproblems and, consequently, expediting the overall training process.

We noticed that in the CG-based classifiers presented in Chapters 3 and 4, most of the training time is spent solving subproblems, making it a bottleneck for the training process. These subproblems are MIPs that only differ in the objective function between CG iterations and are solved sequentially. In Chapter 6, we described our work to exploit similarities in MIPs by reusing the information generated by solving previous instances. We described techniques to reuse primal solutions and branching history. Furthermore, we described techniques to strengthen MIP solvers by extending ideas from reliability branching. Finally, we performed automated parameter tuning and turned off ineffective parts of the MIP solver to reduce solving time. Overall, we demonstrated that these techniques can result in over 20% gains on average in the later instances in the sequence.

7.2 Future Research

In this section, we outline our plan to implement more components in the CG framework to make it even more helpful. These components would directly benefit the work described in Chapters 3 and 4.

When the MP includes integer variables, they can be solved using branch-and-price algorithms. Since branch-and-price can be expensive in terms of solving times, the framework project uses the restricted master heuristic. We plan to implement diving heuristics described in [69] that result in better solutions than the restricted master heuristic.

As discussed in Chapters 4 and 5, some subproblems can be solved in parallel. The framework can benefit from such implementation. This is also planned as a future project in the framework. We also plan to implement more techniques described in Chapter 6 to exploit similarities in the subproblem..

Finally, the CG framework can be easily extended to model CG-based regressors. Furthermore, as described in Chapter 2, CG is used in ML beyond 'predictor' roles (see e.g., [22], [23]). We also plan to extend the CG framework to cover these use cases.

REFERENCES

[1] M. Firat, G. Crognier, A. F. Gabor, C. A. Hurkens, and Y. Zhang, "Column generation based heuristic for learning classification trees," *Computers & Operations Research*, vol. 116, p. 104866, 2020.

[2] E. Danna and C. Le Pape, "Branch-and-price heuristics: A case study on the vehicle routing problem with time windows," in *Column Generation*. Springer, 2005, pp. 99–129.

[3] Z. Jia, Q. Wu, C. Dong, C. Yuen, and Z. Han, "Column generation for optimization problems in communication networks," *IEEE Network*, 2022.

[4] R. Bargetto, T. Garaix, and X. Xie, "A branch-and-price-and-cut algorithm for operating room scheduling under human resource constraints," *Computers & Operations Research*, vol. 152, p. 106136, 2023.

[5] H. Choi, C. Lee, and S. Park, "The column generation approach to the mean-risk model for the portfolio selection problem with spillover risk aversion," *The Engineering Economist*, pp. 1–29, 2023.

[6] L. Xu and S. Haddad Vanier, "Branch-and-price for energy optimization in multi-hop wireless sensor networks," *Networks*, vol. 80, no. 1, pp. 123–148, 2022.

[7] G. Desaulniers, J. Desrosiers, and M. M. Solomon, Eds., *Column Generation*, 1st ed. Springer US, 2005.

[8] G. Desaulniers, J. Desrosiers, and S. Spoorendonk, "Cutting planes for branch-and-price algorithms," *Networks*, vol. 58, pp. 301–310, 12 2011.

[9] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance, "Branch-and-price: Column generation for solving huge integer programs," *Operations Research*, vol. 46, no. 3, pp. 316–329, 1998.

[10] C. Joncour, S. Michel, R. Sadykov, and F. Vanderbeck, "Column generation based primal heuristics," *Electronic Notes in Discrete Mathematics*, vol. 36, pp. 695–702, 2010.

[11] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[12] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*, 1st ed. Springer, 2007.

[13] Y. Bengio, I. Goodfellow, and A. Courville, *Deep learning*. MIT press Cambridge, MA, USA, 2017, vol. 1.

[14] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference and prediction*, 2nd ed. Springer, 2009. [Online]. Available: http://www-stat.stanford.edu/~tibs/ElemStatLearn/

[15] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[16] S. Dash, O. Gunluk, and D. Wei, "Boolean decision rules via column generation," *Advances in neural information processing systems*, vol. 31, 2018.

[17] K. K. Patel, G. Desaulniers, A. Lodi, and F. Lecue, "Explainable prediction of Qcodes for NOTAMs using column generation," *Journal of the Operational Research Society*, vol. 75, no. 1, pp. 173–183, 2024. [Online]. Available: https://doi.org/10.1080/01605682.2023.2181715

[18] K. K. Patel, G. Desaulniers, and A. Lodi, "An improved column-generation-based matheuristic for learning classification trees," *Computers and Operations Research*, vol. 165, p. 106579, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054824000510

[19] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[20] A. Dedieu, R. Mazumder, and H. Wang, "Solving l1-regularized svms and related linear programs: Revisiting the effectiveness of column and constraint generation," pp. 7389–7429, 2022.

[21] A. Demiriz, K. P. Bennett, and J. Shawe-Taylor, "Linear Programming Boosting via Column Generation," *Machine Learning*, vol. 46, no. 1, pp. 225–254, Jan. 2002. [Online]. Available: https://doi.org/10.1023/A:1012470815092

[22] J. Yarkony, Y. Adulyasak, M. Singh, and G. Desaulniers, "Data association via set packing for computer vision applications," *Informs Journal on Optimization*, vol. 2, no. 3, pp. 167–191, 2020.

[23] A. Lodi and J. Ramírez-Ayerbe, "One-for-many counterfactual explanations by column generation," *arXiv preprint arXiv:2402.09473*, 2024.

[24] M. T. Ribeiro, S. Singh, and C. Guestrin, "Model-agnostic interpretability of machine learning," *arXiv preprint arXiv:1606.05386*, 2016.

[25] E. Winter, "The shapley value," *Handbook of game theory with economic applications*, vol. 3, pp. 2025–2054, 2002.

[26] N. Burkart and M. F. Huber, "A survey on the explainability of supervised machine learning," *Journal of Artificial Intelligence Research*, vol. 70, pp. 245–317, 2021.

[27] G. Cervone, P. Franzese, and A. P. Keesee, "Algorithm quasi-optimal (AQ) learning," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 2, pp. 218–236, 2010.

[28] P. Clark and T. Niblett, "The CN2 induction algorithm," *Machine Learning*, vol. 3, no. 4, pp. 261–283, 1989.

[29] P. Clark and R. Boswell, "Rule induction with CN2: Some recent improvements," in *European Working Session on Learning*. Springer, 1991, pp. 151–163.

[30] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

[31] W. W. Cohen, "Fast effective rule induction," in *Machine Learning proceedings 1995*. Elsevier, 1995, pp. 115–123.

[32] M. Muselli and D. Liberati, "Binary rule generation via Hamming Clustering," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 6, pp. 1258–1268, 2002.

[33] J. Fürnkranz, D. Gamberger, and N. Lavrač, *Foundations of rule learning*. Springer Science & Business Media, 2012.

[34] T. Wang, C. Rudin, F. Doshi-Velez, Y. Liu, E. Klampfl, and P. MacNeille, "A bayesian framework for learning rule sets for interpretable classification," *Journal of Machine Learning Research*, vol. 18, no. 70, pp. 1–37, 2017. [Online]. Available: http://jmlr.org/papers/v18/16-003.html

[35] G. Su, D. Wei, K. R. Varshney, and D. M. Malioutov, "Interpretable two-level boolean rule learning for classification," *CoRR*, vol. abs/1606.05798, 2016. [Online]. Available: http://arxiv.org/abs/1606.05798

[36] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques*, 4th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[37] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and regression trees*. Monterey, CA: Wadsworth & Brooks / Cole Advanced Books and Software, 1984.

[38] L. Breiman, "Random forests," *Machine Learning*, vol. 45, pp. 5–32, 10 2001.

[39] C. Lawless, S. Dash, O. Gunluk, and D. Wei, "Interpretable and fair boolean rule sets via column generation," *Journal of Machine Learning Research*, vol. 24, no. 229, pp. 1–50, 2023.

[40] H. Laurent and R. L. Rivest, "Constructing optimal binary decision trees is NP-complete," *Information Processing Letters*, vol. 5, no. 1, pp. 15–17, 1976.

[41] O. Günlük, J. Kalagnanam, M. Li, M. Menickelly, and K. Scheinberg, "Optimal decision trees for categorical data via integer programming," *Journal of Global Optimization*, vol. 81, pp. 233–260, 2021.

[42] S. Verwer and Y. Zhang, "Learning decision trees with flexible constraints and objectives using integer optimization," in *Integration of AI and OR Techniques in Constraint Programming, CPAIOR 2017*, D. Salvagnin and M. Lombardi, Eds. Cham: Springer International Publishing, 2017, pp. 94–103.

[43] ——, "Learning optimal classification trees using a binary linear program formulation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 1625–1632.

[44] D. Bertsimas and J. Dunn, "Optimal classification trees," *Machine Learning*, vol. 106, pp. 1039–1082, 2017.

[45] S. Aghaei, A. Gómez, and P. Vayanos, "Strong optimal classification trees," *arXiv preprint arXiv:2103.15965*, 2021.

[46] E. Carrizosa, C. Molero-Río, and D. Romero Morales, "Mathematical optimization in classification and regression trees," *Top*, vol. 29, no. 1, pp. 5–33, 2021.

[47] F. D'Onofrio, G. Grani, M. Monaci, and L. Palagi, "Margin optimal classification trees," *Computers & Operations Research*, vol. 161, p. 106441, 2024.

[48] V. Blanco, A. Japón, and J. Puerto, "Robust optimal classification trees under noisy labels," *Advances in Data Analysis and Classification*, vol. 16, no. 1, pp. 155–179, 2022.

[49] ——, "Multiclass optimal classification trees with svm-splits," *Machine Learning*, vol. 112, no. 12, pp. 4905–4928, 2023.

[50] R. Blanquero, E. Carrizosa, C. Molero-Río, and D. R. Morales, "Optimal randomized classification trees," *Computers & Operations Research*, vol. 132, p. 105281, 2021.

[51] 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*, 2007, pp. 530–539.

[52] N. Narodytska, A. Ignatiev, F. Pereira, J. Marques-Silva, and I. Ras, "Learning optimal decision trees with SAT." in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, 2018, pp. 1362–1368.

[53] H. Verhaeghe, S. Nijssen, G. Pesant, C.-G. Quimper, and P. Schaus, "Learning optimal decision trees using constraint programming," *Constraints*, vol. 25, pp. 226–250, 2020.

[54] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.

[55] R. Mazumder, S. Wright, and A. Zheng, "Computing Estimators of Dantzig Selector type via Column and Constraint Generation," *arXiv preprint arXiv:1908.06515*, 2019.

[56] P. Belotti, P. Bonami, M. Fischetti, A. Lodi, M. Monaci, A. Nogales-Gómez, and D. Salvagnin, "On handling indicator constraints in mixed integer programming," *Computational Optimization and Applications*, vol. 65, no. 3, pp. 545–566, Dec. 2016. [Online]. Available: https://doi.org/10.1007/s10589-016-9847-8

[57] P. Balamurugan, A. Posinasetty, and S. Shevade, *ADMM for Training Sparse Structural SVMs with Augmented L1 Regularizers*, pp. 684–692. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611974348.77

[58] Y. Nesterov, "Gradient methods for minimizing composite functions," *Mathematical Programming*, vol. 140, no. 1, pp. 125–161, Aug. 2013. [Online]. Available: https://doi.org/10.1007/s10107-012-0629-5

[59] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and*

System Sciences, vol. 55, no. 1, pp. 119–139, 1997. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002200009791504X

[60] G. Rätsch, B. Schölkopf, A. Smola, K.-R. Müller, T. Onoda, and S. Mika, "v-arc: Ensemble learning in the presence of outliers," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12. MIT Press, 2000. [Online]. Available: https://proceedings.neurips.cc/paper/1999/file/28dc6b0e1b33769b4b94685e4f4d1e5c-Paper.pdf

[61] Federal Aviation Administration, "What is a NOTAM," https://www.faa.gov/about/initiatives/notam/what_is_a_notam, 2022.

[62] ——, "International NOTAM (Q) codes," https://www.faa.gov/air_traffic/publications/atpubs/notam_html/appendix_b.html, 2022.

[63] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785

[64] J. Desrosiers and M. E. Lübbecke, *A Primer in Column Generation*. Boston, MA: Springer US, 2005, pp. 1–32.

[65] L. Buitinck *et al.*, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[66] Google, "OR-Tools ," https://developers.google.com/optimization/, 2021.

[67] Gurobi Optimization, "Gurobi Optimizer Reference Manual," https://www.gurobi.com, 2021.

[68] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[69] R. Sadykov, F. Vanderbeck, A. Pessoa, I. Tahiri, and E. Uchoa, "Primal heuristics for branch and price: The assets of diving methods," *INFORMS Journal on Computing*, vol. 31, no. 2, pp. 251–267, 2019.

[70] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[71] A. Gleixner, "Ambros-Gleixner/mipcc23: The MIP workshop 2023 computational competition," 2022. [Online]. Available: https://github.com/ambros-gleixner/MIPcc23

[72] "Developing scikit-learn estimators," https://scikit-learn.org/stable/developers/develop.html, accessed: 2021-11-23.

[73] G. van Rossum, B. Warsaw, and N. Coghlan, "Style guide for Python code," PEP 8, 2001. [Online]. Available: https://www.python.org/dev/peps/pep-0008/

[74] S. Bolusani, M. Besançon, A. Gleixner, T. Berthold, C. d'Ambrosio, G. Muñoz, J. Paat, and D. Thomopulos, "The MIP workshop 2023 computational competition on reoptimization," *arXiv preprint arXiv:2311.14834*, 2023.

[75] S. Bolusani and T. K. Ralphs, "A framework for generalized Benders' decomposition and its application to multilevel optimization," *Mathematical Programming*, vol. 196, no. 1-2, pp. 389–426, 2022.

[76] J. Witzig, "Reoptimization techniques in MIP solvers," *Master's Thesis, TU Berlin*, 2014.

[77] E. Danna, E. Rothberg, and C. L. Pape, "Exploring relaxation induced neighborhoods to improve MIP solutions," *Mathematical Programming*, vol. 102, pp. 71–90, 2005.

[78] T. Berthold, "Primal heuristics for mixed integer programs," Ph.D. dissertation, Zuse Institute Berlin (ZIB), 2006.

[79] G. Gamrath, T. Berthold, S. Heinz, and M. Winkler, "Structure-driven fix-and-propagate heuristics for mixed integer programming," *Mathematical Programming Computation*, vol. 11, no. 4, pp. 675–702, 2019.

[80] D. Thomopulos, W. van Ackooij, C. D'Ambrosio, and M. Stéfanon, "Generating hydro unit commitment instances," *TOP*, pp. 1–31, 2023.

[81] S. S. or an SAP affiliate company, "MILP benchmarks cellphoneco," 2023. [Online]. Available: https://github.com/SAP-samples/ibp-sop-benchmarks-milp-cellphoneco

[82] V. Andréassian, O. Delaigue, C. Perrin, B. Janet, and N. Addor, "CAMELS-FR: A large sample, hydroclimatic dataset for france, to support model testing and evaluation," in *EGU General Assembly Conference Abstracts*, 2021, pp. EGU21–13 349.

[83] M. W. Tanner and L. Ntaimo, "IIS branch-and-cut for joint chance-constrained stochastic programs and application to optimal vaccine allocation," *European Journal of Operational Research*, vol. 207, no. 1, pp. 290–296, 2010.

[84] Cplex, IBM ILOG, "IBM ILOG CPLEX Optimizer," 2023. [Online]. Available: https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-optimizer

[85] G. Ausiello, V. Bonifaci, and B. Escoffier, "Complexity and approximation in reoptimization," in *Computability in Context: Computation and Logic in the Real World*. World Scientific, 2011, pp. 101–129.

[86] B. Schieber, H. Shachnai, G. Tamir, and T. Tamir, "A theory and algorithms for combinatorial reoptimization," *Algorithmica*, vol. 80, pp. 576–607, 2018.

[87] T. Ralphs and M. Güzelsoy, "Duality and warm starting in integer programming," in *The Proceedings of the 2006 NSF Design, Service, and Manufacturing Grantees and Research Conference*, 2006.

[88] G. Gamrath, B. Hiller, and J. Witzig, "Reoptimization techniques for MIP solvers," in *Experimental Algorithms: 14th International Symposium, SEA 2015, Paris, France, June 29–July 1, 2015, Proceedings 14*. Springer, 2015, pp. 181–192.

[89] K. Bestuzheva *et al.*, "The SCIP Optimization Suite 8.0," Zuse Institute Berlin, ZIB-Report 21-41, December 2021. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:0297-zib-85309

[90] C. Blum, P. Pinacho, M. López-Ibáñez, and J. A. Lozano, "Construct, merge, solve & adapt a new general algorithm for combinatorial optimization," *Computers & Operations Research*, vol. 68, pp. 75–88, 2016.

[91] T. Berthold, G. Hendel, and D. Salvagnin, "Transferring information across restarts in mip," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 19th International Conference, CPAIOR 2022, Los Angeles, CA, USA, June 20-23, 2022, Proceedings*. Springer, 2022, pp. 24–33.

[92] T. Achterberg, T. Koch, and A. Martin, "Branching rules revisited," *Operations Research Letters*, vol. 33, no. 1, pp. 42–54, 2005.

[93] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *The journal of Machine Learning research*, vol. 18, no. 1, pp. 6765–6816, 2017.

[94] A. Sabharwal, H. Samulowitz, and C. Reddy, "Guiding combinatorial optimization with UCT," in *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems: 9th International Conference, CPAIOR 2012, Nantes, France, May 28–June1, 2012. Proceedings 9*. Springer, 2012, pp. 356–361.

[95] G. Hendel, "Adaptive large neighborhood search for mixed integer programming," *Mathematical Programming Computation*, pp. 1–37, 2022.

[96] A. Gleixner *et al.*, "MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library," *Mathematical Programming Computation*, vol. 13, no. 3, pp. 443–490, 2021.

[97] G. Angulo, S. Ahmed, and S. S. Dey, "Improving the integer L-shaped method," *INFORMS Journal on Computing*, vol. 28, no. 3, pp. 483–499, 2016.

[98] A. Jiménez-Cordero, J. M. Morales, and S. Pineda, "Warm-starting constraint generation for mixed-integer optimization: A Machine Learning approach," *Knowledge-Based Systems*, vol. 253, p. 109570, 2022.

[99] M. Bénichou, J.-M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent, "Experiments in mixed-integer linear programming," *Mathematical Programming*, vol. 1, pp. 76–94, 1971.

[100] J. T. Linderoth and M. W. Savelsbergh, "A computational study of search strategies for mixed integer programming," *INFORMS Journal on Computing*, vol. 11, no. 2, pp. 173–187, 1999.

[101] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, *Finding cuts in the TSP (A preliminary report)*. Citeseer, 1995, vol. 95.

[102] G. Hendel, "Enhancing MIP branching decisions by using the sample variance of pseudo costs," in *Integration of AI and OR Techniques in Constraint Programming: 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings 12*. Springer, 2015, pp. 199–214.

[103] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Hydra-mip: Automated algorithm configuration and selection for mixed integer programming," in *RCRA workshop*

on experimental evaluation of algorithms for solving problems with combinatorial explosion at the international joint conference on artificial intelligence (IJCAI), 2011, pp. 16–30.

[104] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, pp. 235–256, 2002.

APPENDIX A DETAILED RESULTS FOR PROGRESSIVELY STRENGTHENING AND TUNING MIP SOLVERS FOR REOPTIMIZATION

Table A.1 Effects of providing solution hints on total scores. Best scores are in bold.

Series	Conversion rate (%)	Batch	Total score	
			BASE	SOLHINT
Bound 1	51.0	1-10	**0.78**	0.91
		11-20	**0.77**	0.83
		21-30	**0.72**	0.78
		31-40	**0.80**	0.85
		41-50	**0.60**	0.70
Bound 2	42.9	1-10	**0.76**	0.78
		11-20	0.60	**0.58**
		21-30	**0.64**	0.66
		31-40	0.66	**0.59**
		41-50	**0.63**	0.69
Bound 3	61.2	1-10	0.62	**0.61**
		11-20	0.68	**0.67**
		21-30	0.70	**0.69**
		31-40	0.61	**0.60**
		41-50	0.57	**0.53**
RHS 1	69.4	1-10	**0.53**	0.56
		11-20	0.69	**0.67**
		21-30	0.72	**0.71**
		31-40	0.89	**0.79**
		41-50	**0.90**	0.91
RHS 2	100.0	1-10	0.99	**0.96**
		11-20	0.99	0.99

Table A.1 Effects of providing solution hints on total scores (continued).

		21-30	1.00	1.00
		31-40	1.00	**0.98**
		41-50	0.99	**0.90**
RHS 3	0.0	1-10	0.66	0.66
		11-20	**0.78**	0.79
		21-30	0.71	0.71
		31-40	0.54	0.54
		41-50	0.57	0.57
RHS 4	100.0	1-10	1.00	**0.96**
		11-20	1.00	1.00
		21-30	1.00	**0.99**
		31-40	1.00	**0.98**
		41-50	1.00	**0.99**
Objective 1	100.0	1-10	0.49	**0.48**
		11-20	0.59	**0.55**
		21-30	0.77	**0.75**
		31-40	0.93	**0.88**
		41-50	1.00	**0.95**
Objective 2	100.0	1-10	0.82	0.82
		11-20	**0.97**	1.03
		21-30	**0.97**	1.04
		31-40	0.86	**0.84**
		41-50	**0.93**	0.98
Objective 3	100.0	1-10	0.36	0.36
		11-20	0.37	0.37
		21-30	**0.46**	0.57
		31-40	0.44	**0.38**
		41-50	0.47	**0.45**
RHS Objective 1	93.9	1-10	1.01	**0.91**
		11-20	**0.99**	1.00
		21-30	**0.97**	0.99
		31-40	0.93	**0.92**

Table A.1 Effects of providing solution hints on total scores (continued).

		41-50	1.08	**0.94**
RHS Objective 2	89.8	1-10	0.41	**0.33**
		11-20	**0.46**	0.50
		21-30	**0.23**	0.32
		31-40	0.40	**0.34**
		41-50	0.59	0.59
Matrix 1	100.0	1-10	**0.82**	0.91
		11-20	0.94	**0.92**
		21-30	**0.90**	0.95
		31-40	**0.89**	0.98
		41-50	**0.93**	1.01
Matrix RHS Objective Bound	87.8	1-10	0.60	**0.44**
		11-20	0.55	**0.38**
		21-30	0.50	**0.30**
		31-40	0.36	**0.33**
		41-50	0.51	**0.36**
Matrix RHS Bound	0.0	1-10	0.47	0.47
		11-20	0.66	0.66
		21-30	0.48	0.48
		31-40	0.66	0.66
		41-50	0.59	0.59

Table A.1 Effects of providing solution hints on total scores (end).

Table A.2 Effects of reusing branching history and strong branching for the first instance on total scores. Best scores are in bold.

Series	Batch	Total score		
		BASE	REHIS	REHIS+SB

Table A.2 Effects of reusing branching history and strong branching for the first instance on total scores (continued).

Bound 1	1-10	0.78	**0.62**	0.64
	11-20	0.77	**0.46**	**0.46**
	21-30	0.72	**0.49**	**0.49**
	31-40	0.80	**0.51**	0.57
	41-50	0.60	**0.34**	0.40
Bound 2	1-10	0.76	**0.61**	**0.61**
	11-20	**0.60**	0.61	0.63
	21-30	0.64	**0.53**	0.62
	31-40	0.66	0.60	**0.55**
	41-50	0.63	**0.52**	0.68
Bound 3	1-10	0.62	**0.61**	0.66
	11-20	0.68	**0.67**	**0.67**
	21-30	0.70	0.72	**0.68**
	31-40	0.61	**0.54**	0.63
	41-50	0.57	0.49	**0.44**
RHS 1	1-10	0.53	**0.51**	0.62
	11-20	0.69	**0.64**	0.70
	21-30	0.72	0.77	**0.62**
	31-40	0.89	0.77	**0.71**
	41-50	0.90	0.80	**0.79**
RHS 2	1-10	0.99	0.99	**0.97**
	11-20	0.99	0.99	**0.92**
	21-30	1.00	1.00	**0.97**
	31-40	1.00	0.99	**0.93**
	41-50	0.99	1.00	**0.85**
RHS 3	1-10	0.66	0.62	**0.56**
	11-20	0.78	**0.51**	0.57
	21-30	0.71	0.69	**0.62**
	31-40	0.54	0.59	**0.52**
	41-50	0.57	0.59	**0.52**

Table A.2 Effects of reusing branching history and strong branching for the first instance on total scores (continued).

RHS 4	1-10	1.00	1.00	**0.94**
	11-20	1.00	1.00	**0.94**
	21-30	1.00	1.00	**0.96**
	31-40	1.00	1.00	**0.95**
	41-50	1.00	1.00	**0.92**
Objective 1	1-10	**0.49**	**0.49**	0.52
	11-20	0.59	**0.57**	**0.57**
	21-30	**0.77**	**0.77**	0.78
	31-40	0.93	0.89	**0.87**
	41-50	1.00	0.97	**0.92**
Objective 2	1-10	**0.82**	0.92	1.08
	11-20	**0.97**	1.05	1.05
	21-30	0.97	**0.95**	0.96
	31-40	0.86	**0.81**	0.85
	41-50	**0.93**	1.10	1.12
Objective 3	1-10	**0.36**	0.38	0.59
	11-20	**0.37**	0.57	0.54
	21-30	0.46	0.56	**0.44**
	31-40	**0.44**	0.48	**0.44**
	41-50	0.47	**0.37**	0.42
RHS Objective 1	1-10	1.01	**0.96**	1.08
	11-20	**0.99**	1.02	1.04
	21-30	**0.97**	1.00	1.01
	31-40	**0.93**	0.99	**0.93**
	41-50	1.08	**1.00**	**1.00**
RHS Objective 2	1-10	**0.41**	0.42	0.47
	11-20	**0.46**	0.48	0.49
	21-30	**0.23**	0.37	0.24
	31-40	**0.40**	0.44	**0.40**
	41-50	0.59	0.58	**0.56**

Table A.2 Effects of reusing branching history and strong branching for the first instance on total scores (continued).

Matrix 1	1-10	**0.81**	0.87	0.90
	11-20	0.94	**0.85**	0.97
	21-30	**0.90**	0.96	0.98
	31-40	**0.89**	0.98	0.94
	41-50	0.93	0.89	**0.87**
Matrix RHS Objective Bound	1-10	**0.60**	0.68	0.75
	11-20	**0.55**	0.63	0.60
	21-30	**0.50**	0.65	0.63
	31-40	**0.36**	0.46	0.43
	41-50	**0.51**	0.59	0.54
Matrix RHS Bound	1-10	**0.47**	0.59	0.53
	11-20	0.67	**0.54**	0.71
	21-30	0.48	0.57	**0.45**
	31-40	0.66	**0.53**	0.60
	41-50	0.60	0.67	**0.55**

Table A.2 Effects of reusing branching history and strong branching for the first instance on total scores (end).

Table A.3 Effects of automated parameter tuning on total scores. Best scores are in bold.

Series	Batch	Total scores	
		NOTUNING	TUNING
Bound 1	1-10	**0.73**	0.85
	11-20	0.61	**0.50**
	21-30	0.71	**0.45**
	31-40	0.84	**0.44**
	41-50	0.61	**0.32**

Table A.3 Effects of automated parameter tuning on total scores (continued).

	1-10	**0.57**	0.58
	11-20	**0.45**	0.50
Bound 2	21-30	**0.40**	0.43
	31-40	0.40	**0.39**
	41-50	0.40	0.40
	1-10	**0.54**	0.58
	11-20	**0.48**	0.62
Bound 3	21-30	**0.52**	0.64
	31-40	**0.49**	0.50
	41-50	**0.39**	0.42
	1-10	**0.50**	0.77
	11-20	**0.64**	0.68
RHS 1	21-30	0.71	**0.61**
	31-40	0.72	**0.69**
	41-50	0.79	**0.48**
	1-10	**0.66**	0.68
	11-20	0.63	**0.55**
RHS 2	21-30	0.70	**0.57**
	31-40	0.67	**0.59**
	41-50	0.66	**0.52**
	1-10	**0.61**	0.79
	11-20	**0.75**	0.85
RHS 3	21-30	0.57	**0.55**
	31-40	**0.52**	0.57
	41-50	0.53	**0.37**
	1-10	0.71	**0.69**
	11-20	0.64	**0.58**
RHS 4	21-30	0.70	**0.54**
	31-40	0.67	**0.56**
	41-50	0.67	**0.56**
	1-10	**0.51**	0.52

Objective 1

Table A.3 Effects of automated parameter tuning on total scores (continued).

	11-20	0.56	**0.49**
	21-30	0.78	**0.60**
	31-40	0.89	**0.74**
	41-50	0.98	**0.96**
Objective 2	1-10	0.97	**0.93**
	11-20	**1.09**	1.20
	21-30	1.11	**1.07**
	31-40	**0.80**	0.91
	41-50	1.07	**1.04**
Objective 3	1-10	0.46	**0.38**
	11-20	0.63	**0.37**
	21-30	0.64	**0.31**
	31-40	0.79	**0.45**
	41-50	0.76	**0.50**
RHS Objective 1	1-10	1.07	**1.03**
	11-20	1.00	**0.98**
	21-30	1.00	**0.89**
	31-40	0.93	0.93
	41-50	0.96	**0.83**
RHS Objective 2	1-10	**0.49**	0.77
	11-20	**0.59**	0.77
	21-30	**0.53**	0.62
	31-40	**0.59**	0.69
	41-50	**0.66**	0.80
Matrix 1	1-10	**1.00**	1.04
	11-20	**0.95**	1.02
	21-30	**1.00**	1.03
	31-40	**1.05**	1.06
	41-50	1.08	**1.06**
Matrix RHS Objective Bound	1-10	**0.63**	0.66
	11-20	0.48	**0.44**
	21-30	0.30	**0.27**

Table A.3 Effects of automated parameter tuning on total scores (continued).

	31-40	0.42	**0.33**
	41-50	0.50	**0.30**
Matrix RHS Bound	1-10	**0.61**	0.91
	11-20	**0.64**	0.75
	21-30	0.75	0.75
	31-40	0.90	**0.77**
	41-50	0.73	**0.69**

Table A.3 Effects of automated parameter tuning on total scores (end).

Table A.4 Effects of turning off inefficient solver parts on total scores. Best scores are in bold.

Series	Batch	Total scores	
		NOTURNOFF	TURNOFF
Bound 1	1-10	0.78	0.78
	11-20	0.48	**0.46**
	21-30	0.46	**0.42**
	31-40	0.48	**0.39**
	41-50	**0.34**	0.35
Bound 2	1-10	0.59	0.59
	11-20	**0.43**	0.50
	21-30	**0.38**	0.42
	31-40	0.36	0.36
	41-50	0.41	0.41
Bound 3	1-10	0.58	0.58
	11-20	0.59	0.59
	21-30	**0.54**	0.56
	31-40	**0.49**	0.52
	41-50	**0.40**	0.42

Table A.4 Effects of turning off inefficient solver parts on total scores (continued).

RHS 1	1-10	0.76	0.76
	11-20	0.71	**0.62**
	21-30	**0.55**	0.63
	31-40	**0.62**	0.63
	41-50	0.66	**0.54**
RHS 2	1-10	0.66	0.66
	11-20	0.54	0.54
	21-30	0.59	**0.42**
	31-40	0.55	**0.27**
	41-50	0.51	**0.26**
RHS 3	1-10	0.62	0.62
	11-20	**0.72**	0.82
	21-30	**0.50**	0.65
	31-40	**0.62**	0.79
	41-50	**0.46**	0.51
RHS 4	1-10	0.70	0.70
	11-20	0.57	0.57
	21-30	0.53	**0.42**
	31-40	0.56	**0.30**
	41-50	0.55	**0.29**
Objective 1	1-10	0.48	0.48
	11-20	0.45	**0.43**
	21-30	0.67	**0.64**
	31-40	0.74	**0.72**
	41-50	0.90	**0.88**
Objective 2	1-10	0.93	0.93
	11-20	1.29	**1.18**
	21-30	1.00	**0.95**
	31-40	**0.94**	0.95
	41-50	**1.07**	1.13
Objective 3	1-10	0.43	0.43

Table A.4 Effects of turning off inefficient solver parts on total scores (continued).

	11-20	**0.39**	0.49
	21-30	**0.40**	0.46
	31-40	0.41	**0.35**
	41-50	**0.36**	0.42
RHS Objective 1	1-10	1.03	1.03
	11-20	1.03	**1.00**
	21-30	**0.89**	0.95
	31-40	**0.72**	0.86
	41-50	0.82	0.82
RHS Objective 2	1-10	0.76	0.76
	11-20	**0.63**	0.74
	21-30	0.69	**0.56**
	31-40	0.63	**0.50**
	41-50	0.72	**0.68**
Matrix 1	1-10	1.04	1.04
	11-20	**0.97**	1.04
	21-30	1.05	1.05
	31-40	**1.06**	1.07
	41-50	1.05	**1.04**
Matrix RHS Objective Bound	1-10	0.58	0.58
	11-20	0.35	0.35
	21-30	**0.25**	0.28
	31-40	**0.32**	0.33
	41-50	0.30	0.30
Matrix RHS Bound	1-10	0.80	0.80
	11-20	0.78	0.78
	21-30	0.67	**0.61**
	31-40	0.93	**0.86**
	41-50	**0.65**	0.76

Table A.4 Effects of turning off inefficient solver parts on total scores (end).

We present the aggregated statistics for techniques presented in this paper in Table A.5 (grouped by changing component), Table A.6 (grouped by geometric mean of solve time in BASE), and Table A.7 (grouped by BASE score). Each table shows the aggregated BASE total score and percentage improvements caused by each technique. Since the number of series for each group is too small, we advise readers to be cautious while making generalized conclusions based on these tables (e.g., TURNOFF works best when RHS is changing).

Table A.5 Aggregated results for changing components.

Series	Base score	SOLHINT (%)	REHIS (%)	REHIS+SBTUNING (%)	TURNOFF (%)
Bound only	0.676	-2.9	17.18	13.16	3.07	-0.75
All Bound	0.621	3.9	6.28	4.76	3.35	-0.79
RHS only	0.848	1.64	3.6	8.66	6.19	8.44
All RHS	0.759	5.66	-1.89	1.75	1.34	4.23
Objective only	0.695	-0.15	-5.38	-7.92	15.85	-1.18
All Objective	0.667	5.44	-7.49	-7.63	6.69	-0.67
Matrix only	0.896	-6.47	-1.79	-4.25	-2.56	-1.35
All Matrix	0.657	7.23	-7.31	-6.64	1.66	-1.02

Table A.6 Aggregated results for series grouped by geometric means of solving time in BASE.

Time limit level	Base score	SOLHINT (%)	REHIS (%)	REHIS+SBTUNING (%)	TURNOFF (%)
1 (0s - 100s)	0.729	8.27	-7.25	-1.99	3.11	12.38
2 (100s - 200s)	0.539	-0.87	0.19	-4.82	17.56	-6.55
3 (200s - 300s)	0.781	-1.89	-0.55	-1.57	-1.87	1.13
4 (300s - 400s)	0.681	2.24	5.02	6.91	-2.69	-5.34
5 (400s - 500s)	0.865	-3.24	17.13	14.32	16.45	0.86

Table A.7 Aggregated results for series grouped by average BASE scores.

Base score level	Base score	SOLHINT (%)	REHIS (%)	REHIS+SBTUNING (%)	TURNOFF (%)
1 (0-0.6)	0.479	6.81	-10.52	-8.68	4.66	-1.05
2 (0.6-0.7)	0.649	0.64	8.49	7.88	-7.56	-7.95
3 (0.7-0.8)	0.745	-1.33	14.29	13.73	13.92	3.98
4 (0.8-0.9)	0.896	-6.47	-1.79	-4.25	-2.56	-1.35
5 (0.9-1)	0.975	1.33	-1.49	-0.09	7.45	11.04