CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Machine learning for structural reasoning in Boolean Satisfiability
Authors	Dalla, Marco
Publication date	2024
Original Citation	Dalla, M. 2024. Machine learning for structural reasoning in Boolean Satisfiability. PhD Thesis, University College Cork.
Type of publication	Doctoral thesis
Rights	© 2024, Marco Dalla. - https://creativecommons.org/publicdomain/zero/1.0/
Download date	2026-11-15 14:47:50
Item downloaded from	https://hdl.handle.net/10468/17933



UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Machine Learning for Structural Reasoning in Boolean Satisfiability

Marco Dalla

Thesis submitted for the degree of Doctor of Philosophy



NATIONAL UNIVERSITY OF IRELAND, CORK

COLLEGE OF SCIENCE, ENGINEERING AND FOOD SCIENCE

SCHOOL OF COMPUTER SCIENCE & INFORMATION TECHNOLOGY
INSIGHT SFI RESEARCH CENTRE FOR DATA ANALYTICS

December 2024

Head of School: Prof. Dirk Pesch

Supervisors: Prof. Barry O' Sullivan
Dr. Andrea Visentin

Contents

List of Figures vi
List of Tables ix
Acknowledgements xii
Abstract xiii

1 Introduction **1**
1.1 Motivation 1
1.2 A Brief History of SAT 2
1.3 Thesis Statement 3
1.4 Contributions 4
1.5 Publications 6
1.6 Thesis Structure 7

2 Background **9**
2.1 Artificial Intelligence 9
2.1.1 Definitions and Perspectives 9
2.1.2 Current Developments in AI 10
2.1.3 Challenges and Ethical Considerations 11
2.1.4 Algorithms in AI 12
2.2 Boolean Satisfiability 12
2.2.1 Model Counting 13
2.2.2 Conjunctive Normal Form 14
2.2.3 The DPLL Procedure 15
2.2.4 Lookahead 16
2.2.5 Conflict-Driven Clause Learning 18
2.2.5.1 Conflict Analysis and Clause Resolution 20
2.2.5.2 Fast Conflict Creation 21
2.2.5.3 CDCL Branching Techniques 21
2.2.5.4 CDCL Polarity Heuristics 22
2.2.5.5 Metrics Related to Conflicts 22
2.2.5.6 Leading CDCL Solver Systems 23
2.2.6 Local Search Solvers in SAT 23
2.2.6.1 Basic Concepts 24
2.2.6.2 Main Algorithms 25
2.2.6.3 Application of Local Search to SAT 26
2.2.7 State of the Art in SAT Solving 26
2.2.7.1 Improvements in Conflict-Driven Clause Learning 26
2.2.7.2 Clause Management and Simplification 27
2.2.7.3 Restart Strategies and Incremental Solving . . 29
2.3 Machine Learning 29
2.3.1 Supervised Learning 29
2.3.2 Unsupervised Learning 30
2.3.3 Reinforcement Learning 31
2.3.4 Deep Learning 31
2.3.4.1 Deep Neural Networks 31

2.3.4.2 The Backpropagation Algorithm 32
2.3.4.3 Advanced Techniques in Deep Learning 33
2.3.4.4 Challenges and Future Directions 35
2.4 Machine Learning-Based SAT Solving 35
2.4.1 SAT Solver Augmentation 36
2.4.1.1 Variable Selection 36
2.4.1.2 Heuristic Learning 36
2.4.1.3 Clause Management 37
2.4.1.4 Configuration Tuning. 37
2.4.1.5 Impact of Machine Learning on SAT Solvers . . 37
2.4.2 Deep Learning and SAT 38
2.4.2.1 End-to-End Learning for SAT Solving 38
2.4.2.2 Graph Neural Networks in SAT Solving 39
2.4.2.3 Neural Networks as Standalone SAT Solvers . . 39
2.4.2.4 Innovative Neural Network Architectures for SAT 39
2.4.3 Challenges and Future Directions 40

3 Machine Learning Predictions based on SAT Features 41
3.1 Introduction 41
3.2 Dataset Description 43
3.2.1 Dataset for Satisfiability and Category Prediction 43
3.2.1.1 Problem Families 44
3.2.1.2 Instance Distribution and Characteristics . . . 46
3.2.2 Dataset for Model Counting 46
3.2.2.1 Dataset 1 47
3.2.2.2 Dataset 2 47
3.2.2.3 Dataset 3 47
3.2.2.4 Dataset 4 48
3.3 Feature Sets 49
3.3.1 SATzilla Features 50
3.3.1.1 Problem Size 51
3.3.1.2 Variable-Clause Graph (VCG) 52
3.3.1.3 Variable Graph (VG) 52
3.3.1.4 Balance Features 53
3.3.1.5 Bias Measurement 53
3.3.1.6 Proximity to Horn Formula 53
3.3.1.7 DPLL Probing 54
3.3.1.8 Local Search Probing 54
3.3.2 The Ansotegui et al. Features 55
3.3.2.1 Power Law Exponent 55
3.3.2.2 Modularity 55
3.3.2.3 Fractal Dimension 56
3.3.3 The Alfonso et al. Features 56
3.3.3.1 Clause Variable Graphs (CV+ and CV-) 56
3.3.3.2 Variable Graph (VG) 56
3.3.3.3 Clause Graph (CG) 57

3.3.3.4 Resolution Graph (RG) . . . 57
3.3.3.5 Binary Implication Graph (BIG) . . . 57
3.3.3.6 AND/BAND/EXO Gate Graphs . . . 57
3.3.3.7 Recursive Weight Heuristic . . . 58
3.3.4 SATfeatPy: A Unified Feature Extraction Framework . . 58
3.3.5 Computational Overhead and Benefits of Feature Extraction . . . 59
3.4 Method . . . 60
3.4.1 Algorithms . . . 60
3.4.1.1 Machine Learning . . . 60
3.4.1.2 Random Forest . . . 60
3.4.1.3 XGBoost . . . 61
3.4.1.4 Model Counters . . . 62
3.4.2 Metrics . . . 64
3.5 Evaluation . . . 66
3.5.1 Feature Time analysis . . . 66
3.5.2 SAT/UNSAT Classification . . . 67
3.5.3 Problem Category Classification . . . 69
3.5.4 Solver Selection Based on Instance Type . . . 71
3.5.5 Model Counting . . . 71
3.5.5.1 Accuracy of the Approximation . . . 72
3.5.5.2 Computational Time . . . 77
3.5.5.3 Feature Importance . . . 78
3.6 Model Count Guarantees . . . 82
3.7 Conclusions . . . 82

4 Deep Learning Predictions **84**
4.1 Introduction . . . 84
4.2 Encoding Techniques for SAT Problem Representation in Deep Learning Models . . . 87
4.2.1 Binary Encoding . . . 87
4.2.2 Image-based Encoding . . . 89
4.2.3 Graph-based Encoding . . . 91
4.2.4 Efficiency of Deep Learning Encodings . . . 92
4.3 Method . . . 93
4.3.1 Training Approach . . . 93
4.3.2 Data Preparation . . . 93
4.3.2.1 Dataset Description . . . 93
4.3.2.2 Preprocessing . . . 94
4.3.3 Model Architectures . . . 94
4.3.3.1 Overview of Models . . . 94
4.3.3.2 Direct Approaches . . . 94
4.3.3.3 Feature Extraction Methods . . . 100
4.3.4 Training and Hyperparametrization . . . 104
4.3.4.1 CNN Models . . . 105
4.3.4.2 Convolutional Autoencoder (CAE) . . . 106

4.3.4.3 Graph Neural Network (GNN) 107
4.3.4.4 Graph Variational Autoencoder (GVAE) 107
4.4 Evaluation 108
4.4.1 SAT/UNSAT Prediction 108
4.4.2 Comparison between automatically extracted features and handcrafted features 109
4.4.3 Instance Category Prediction 109
4.4.4 Model Count Prediction 111
4.4.5 Analysis and Discussion 112
4.5 Conclusions 113

5 SAT Generation with Deep Learning Models 114
5.1 Introduction 114
5.2 Categories of SAT Problems 117
5.2.1 Random and Crafted SAT Problem Generation 117
5.2.2 Industrial SAT Problem Generation 117
5.2.3 Deep Learning Approaches to SAT Problem Generation . 118
5.3 Method 118
5.3.1 Data Preparation 119
5.3.1.1 Preprocessing 119
5.3.1.2 Postprocessing 119
5.3.2 Model Architecture 120
5.3.3 Training and Generation 123
5.3.4 Competitors 124
5.3.4.1 EGNN2S and G2SAT 124
5.3.4.2 W2SAT 126
5.3.5 Metrics 126
5.3.5.1 Modularity and Clustering Coefficient 126
5.3.5.2 Ranking Coefficients 128
5.3.5.3 SATzilla Feature Distance 129
5.4 Evaluation 129
5.4.1 Solver Performance 133
5.4.2 Generation Computational Times 135
5.4.3 SATzilla Feature Distance 135
5.5 Conclusions 137

6 Conclusions and Future Works 138
6.1 Conclusions 138
6.2 Future Work 140
6.2.1 Improved Variable Selection and Branching Heuristics . 140
6.2.2 Deep Learning-Driven SAT Solvers 140
6.2.3 Advancements in SAT Problem Generation 141
6.2.4 Scalability and Generalization of Machine Learning Models 141
6.2.5 Explainability and Interpretability in SAT Solvers 141
6.2.6 Refinement of Latent Representations 141
6.2.7 Hybrid Feature Sets and Novel Encodings 142
6.2.8 Integration of Generative Models with Solvers 142

6.2.9 Broader Applications of Generative Models 142
6.2.10 Algorithm Portfolios for Model Counting 142

List of Figures

2.1 Execution of the DPLL procedure. 17
2.2 Example of an implication graph 19

3.1 Histogram of number of variables for the dataset 44
3.2 Histograms of number of clauses for the dataset 45
3.3 Natural logarithm distribution of the number of solutions for the instances of Dataset 1 48
3.4 Natural logarithm distribution of the number of solutions for the instances of Dataset 2 49
3.5 Natural logarithm distribution of the number of solutions for the instances of Dataset 3 50
3.6 Natural logarithm distribution of the number of solutions for the instances of Dataset 4 51
3.7 Example of a more complex Variable-Clause Graph (VCG). Circles represent variables and rectangles represent clauses. 52
3.8 Example of a Variable Graph (VG) representation. Nodes represent variables and edges represent co-occurrences in clauses. . . 53
3.9 Example of a Variable Incidence Graph (VIG) representation with high modularity. Solid lines indicate strong internal connections within modules, while dashed lines indicate sparser connections between modules. 55
3.10 Example of a Clause Variable Graph (CVG) representation. Circles represent literals and rectangles represent clauses. 56
3.11 Example of a Clause Graph (CG) representation. Nodes represent clauses and edges represent shared literals between clauses. 57
3.12 Example of a Resolution Graph (RG) representation. Nodes represent clauses and edges represent resolution paths. 57
3.13 Example of a Binary Implication Graph (BIG) representation. Nodes represent literals and directed edges represent implications. 58
3.14 Example of an AND Gate Graph representation. Nodes represent literals and gates, and edges represent connections. 58
3.15 Random Forest Scheme [Tse21] 61
3.16 Time to generate features vs. size. The mean time to solve each set of instances is plotted in a line, and the variance is shown in the error bars. 66
3.17 SHAP analysis for the RFC on the SAT/UNSAT classification task. The eight most important features, primarily derived from local search probing, highlight the importance of empirical hardness metrics. 69

3.18 SHAP analysis for the RFC on the problem category classification task. The most relevant features are derived from graph-based representations, such as the Binary Implication Graph and AND-gate graph, highlighting their importance for capturing structural characteristics. 71
3.19 Plot of GANAK solutions, ApproxMC approximate solutions ($\delta = 0.2, 1$), ApproxMC guarantee interval, and Random Forest Regression predicted solutions for Dataset 1 testing instances. The decimal logarithm of the number of solutions (Y-axis) for each instance (X-axis) is plotted in descending order. 75
3.20 Plot of GANAK solutions, ApproxMC approximate solutions ($\delta = 0.2, 1$), ApproxMC guarantee interval, and Random Forest Regression predicted solutions for Dataset 2 testing instances. The decimal logarithm of the number of solutions (Y-axis) for each instance (X-axis) is plotted in descending order. 75
3.21 Plot of GANAK solutions, ApproxMC approximate solutions ($\delta = 0.2, 1$), and Random Forest Regression predicted solutions for Dataset 3 testing instances. The decimal logarithm of the number of solutions (Y-axis) for each instance (X-axis) is plotted in descending order. 76
3.22 Plot of GANAK solutions, ApproxMC approximate solutions ($\delta = 0.2, 1$), and Random Forest Regression predicted solutions for Dataset 4 testing instances. The decimal logarithm of the number of solutions (Y-axis) for each instance (X-axis) is plotted in descending order. 76
3.23 SHAP analysis performed for the Random Forest Regressor on Dataset 1 testing set. 78
3.24 SHAP analysis performed for the Random Forest Regressor on Dataset 2 testing set. 79
3.25 SHAP analysis performed for the Random Forest Regressor on Dataset 3. 79
3.26 SHAP analysis performed for the Random Forest Regressor on Dataset 4. 80

4.1 Image extracted from SAT problem instance. 90
4.2 Convolutional Neural Network architecture. 96
4.3 Convolutional Neural Network architecture for image-encoded SAT instances. 98
4.4 Graph Neural Network (GNN) architecture. The output layers are adapted for binary classification (SAT/UNSAT), category classification, and regression tasks. 100
4.5 Convolutional Autoencoder architecture. The decoder mirrors the encoder structure with transposed convolutions, reconstructing the input from the latent space representation z. 102
4.6 Graph Variational Autoencoder (GVAE) architecture. 104
4.7 SAT/UNSAT classification accuracy. 110

5.1 Variational Graph Autoencoder consisting of an encoder (right) and decoder (left). The encoder features 2 graph convolutional layers, followed by mean aggregation, and 2 fully connected layers for mean and standard deviation (μ and σ) used to parameterize the latent variable $\mathbf{z}$. The decoder mirrors the encoder structure, with dense layers and appropriate activation, dropout, and normalization layers. 122
5.2 Diagram of the pipeline used for GVAE2SAT. 125
5.3 Distribution of the test instances 130
5.4 LIG - Average Clustering Coefficient 132
5.5 VIG - Average Clustering Coefficient 132
5.6 LIG - Modularity 132
5.7 LCG - Modularity 132
5.8 VCG - Modularity 132
5.9 VIG - Modularity 132
5.10 Modularity and clustering coefficient for the graphs of the generated instances. 132

List of Tables

3.1 Statistics for Dataset 1 and Dataset 2. The value -1 indicates a #SAT problem without solutions, i.e., an unsatisfiable instance . 49
3.2 Statistics for Dataset 3 and Dataset 4. The value -1 indicates a #SAT problem without solutions, i.e., an unsatisfiable instance . 50
3.3 SAT/UNSAT classification accuracies and average computational times (in seconds) across feature sets. 67
3.4 Problem category classification accuracies and computational times across feature sets for both Random Forest and XGBoost classifiers. 69
3.5 Accuracy comparison for Dataset 1. For this table, only 429 instances out of 450 are considered, as those are the only ones where ApproxMC and GANAK are able to provide a count within the 5000 seconds timeout. 73
3.6 Accuracy comparison for Dataset 2. 73
3.7 Accuracy comparison for Dataset 3. For this table, only 993 instances out of 1597 are considered, as those are the only ones where ApproxMC and GANAK are able to provide a count within the 5000 seconds timeout. 74
3.8 Evaluation metrics comparison for Dataset 4, including the 12instances where ApproxMC and GANAK did not provide a model count within a 5000 seconds timeout (Dataset 3 Extra). 74
3.9 Comparison of computational times for Dataset 1 and Dataset 2. For Dataset 1, ApproxMC ($\delta = 0.2$) has a maximum runtime of 3324 seconds, with GANAK reaching up to 4958 seconds. Dataset 2 results are significantly faster, with ApproxMC and GANAK completing within less than a second. Times are reported for both training and testing phases across different methods. 77
3.10 Comparison of computational times for Dataset 3 and Dataset 4. For Dataset 3, ApproxMC ($\delta = 0.2$) reaches the timeout of 5000 seconds in over 204 instances, and GANAK in 201 instances. For Dataset 4, counters reach the timeout for 12 instances using ApproxMC and GANAK. Times are reported for training and testing phases across all methods. 78

4.1 SAT/UNSAT accuracy prediction for the DL algorithm compared with the ML approach 109
4.2 Instance classification accuracy 111
4.3 Model Counting Regression accuracy 112

5.1 Modularity and clustering coefficient for the graphs of the generated instances 131
5.2 Percentage of satisfiability of generations 133

5.3 Average Spearman and Kenadall ranking metrics of the solving times for the generated instances compared to the solving times for the new instances. The computational time is provided by PySat; the solvers' initialization clause load is not considered. . 134
5.4 Running Times (s) 135
5.5 Error of SATzilla Features Reduced with t-SNE 136

I, Marco Dalla, certify that this thesis is my own work and has not been submitted for another degree at University College Cork or elsewhere.

Marco Dalla

Acknowledgements

This thesis has emanated from research conducted with financial support of Science Foundation Ireland under Grant 16/RC/3918, 12/RC/2289-P2, and 18/CRT/6223, which are co-funded under the European Regional Development Fund.

This thesis wouldn't exist without the many people who helped me get here.

First, my supervisors. Prof Barry O'Sullivan has been everything you could want in a supervisor: always ready with encouragement, knowledge, and practical advice. He pushed me to sharpen every argument while keeping me focused on what really mattered. Dr Andrea Visentin started as a good friend and became my co-supervisor when he joined in my second year. His willingness to tackle problems at any hour, paired with his calm "we'll figure this out" approach, got me through the toughest parts.

My family is everything. My parents Anna and Roberto, and my grandmother Bruna, never stopped believing in me, even when I wasn't sure I believed in myself. They kept me grounded with their humour, reassurance, and gentle reminders that the hard work would pay off.

Emma, I can't thank you enough. You were the voice that talked me down from panic, the quiet company during endless late nights, and the first person I wanted to see when I finally looked up from my computer. You somehow knew exactly when to be patient and when to drag me away from work for a walk, a proper meal, or a movie (despite my protests that I was "almost finished"). You kept me sane. I'm also extremely grateful to Emma's parents, Brendan and Lorraine, and Emma's whole extended family - cousins, aunts, uncles, and nans - for making me feel like part of the family and giving me a real home away from home whenever I needed it.

Having friends who get what research is like makes all the difference. Diego, Federico, and Andrea Rossi, thanks for the advice, the much needed coffee breaks, and the card games. Your honesty and humour turned what could have been overwhelming moments into something I could actually handle. And to everyone else who read drafts, shared data, asked the right questions, or just checked in when I needed it most, thank you. Every contribution, big or small, helped shape this work. I'm truly grateful to all of you.

Abstract

The intersection of machine learning and propositional Boolean Satisfiability (SAT) offers transformative possibilities for solving some of the most challenging computational problems. This thesis investigates the use of modern machine learning (ML) and deep learning (DL) methodologies to enhance Boolean Satisfiability and Model Counting. Building upon the foundational understanding of SAT and its role as an NP-complete problem, this work addresses challenges related to structural reasoning, satisfiability and model counting prediction, feature extraction and instance generation. The contributions of this research are varied. First, by leveraging a diverse array of features and representations, we develop machine learning algorithms for SAT/UNSAT classification, problem categorization, and approximate model counting. These models demonstrate predictive accuracy and superior computational efficiency compared to traditional handcrafted heuristics. Second, we automate these procedures through the deployment of deep learning algorithms, significantly reducing dependency on manual engineering while improving adaptability to diverse SAT instances. Finally, we extend the application of the deep learning architectures to SAT instance generation. These approaches enable the generation of structurally diverse, statistically realistic SAT instances that serve as robust benchmarks for solver evaluation. The introduction of novel evaluation metrics ensures the practical utility of these generative models, emphasizing their contributions to advancing SAT-solving strategies. Through an extensive comparative analysis of traditional and machine learning driver SAT-analysis methodologies, this thesis extends the body of work that focuses on the integration of these paradigms. Its findings contribute to the broader field of computational logic, offering insights into scalable and interpretable solutions for Boolean reasoning tasks.

Chapter 1

Introduction

1.1 Motivation

Boolean Satisfiability (SAT) is a foundational problem in computer science, known for being the first NP-complete problem [Coo71]. This classification has driven extensive research in theoretical computer science, particularly in areas of algorithmic complexity and optimization [AS09]. SAT and its extension, Model Counting (#SAT), have significant applications in real-world domains such as verification, planning, and combinatorial optimization [GZL+23]. In verification, SAT solvers are crucial for checking the correctness of hardware and software systems, helping identify flaws in designs before implementation. Similarly, SAT models are applied to complex planning problems, where the goal is to satisfy multiple constraints in a feasible manner, and in combinatorial optimization problems like scheduling and resource allocation, where they assist in identifying optimal solutions. Moreover, tasks like model counting, which quantify the number of satisfying assignments, are essential for probabilistic inference, planning under uncertainty, and model verification in complex systems [GZL+23].

As problem instances grow more complex, especially in fields like artificial intelligence, cybersecurity, and circuit design, the need for faster and more scalable SAT solving techniques becomes more critical [AS09]. Classical algorithms like DPLL and CDCL have achieved remarkable progress in solving SAT instances efficiently [DP60]. However, their reliance on handcrafted heuristics and the exponential growth of problem space in worst-case scenarios pose scalability challenges [ZMMM].

In recent years, machine learning (ML) and deep learning (DL) have emerged as powerful tools to address these limitations [GZL+23]. ML and DL techniques offer the potential to automatically extract relevant features from SAT instances, predict satisfiability, improve solver performance, and even generate novel problem instances for testing and benchmarking solvers. These advancements open new possibilities for enhancing both the accuracy and speed of SAT solvers across various applications, furthering their practical use in industry and research.

This thesis seeks to contribute to this growing body of research by focusing on leveraging machine learning and deep learning for SAT problem-solving, including the automation of feature extraction, prediction tasks, and the generation of realistic SAT instances, all aimed at improving solver efficiency and performance.

1.2 A Brief History of SAT

The Boolean Satisfiability Problem (SAT) is one of computer science's oldest problems and a cornerstone of computational complexity theory. Its origins are traced to the foundation of Boolean Algebra laid out by mathematician and logician George Boole [Coo71]. Boolean Algebra is a mathematical system representing logic through variables that take true or false values. This abstraction made it possible to formalize logical reasoning in mathematical terms, paving the way for SAT as a formal computational problem.

SAT was first explicitly formulated as a decision problem: given a Boolean formula, can we determine whether there exists an assignment of true or false values to its variables that makes the formula evaluate to true? While this question is deceptively simple, it is at the heart of some of the most profound challenges in computer science. SAT became famous when Stephen Cook introduced it in 1971, and it was the first problem proven to be NP-complete in his seminal paper "The Complexity of Theorem-Proving Procedures" [Coo71]. This classification means that if we can find an efficient algorithm to solve SAT, we can solve all other problems in the NP class efficiently—a tantalizing yet unsolved prospect.

In the following decades, SAT emerged as a central problem in theoretical and applied computer science. Early work on SAT solvers began in the 1950s and 1960s with methods like the Davis-Putnam algorithm and its successor, the

DPLL algorithm [DP60]. These algorithms are at the foundations of SAT solving, as they are the first to introduce systematic ways to explore possible assignments while pruning the search space effectively. The introduction of the Conflict-Driven Clause Learning (CDCL) paradigm in the 1990s marked a significant leap in solver performance, making SAT solvers practical for large-scale real-world applications [ZMMM].

SAT solving has practical relevance in many domains. In hardware and software verification, SAT solvers check the correctness of circuits and programs by ensuring that specific undesirable configurations are unsatisfiable [GZL+23]. In artificial intelligence and planning, SAT is used to model and solve problems involving constraints, such as scheduling tasks or routing logistics. Moreover, SAT has other applications in cryptography, bioinformatics, and even philosophy, where it can formalize and analyze logical arguments [GZL+23].

Despite its successes, SAT remains a challenging problem. While modern solvers can handle instances with millions of variables in some cases, the problem's NP-complete nature ensures that certain instances remain intractable. Researchers continue to explore ways to extend the capabilities of SAT solvers, including leveraging advances in machine learning and deep learning, which is the focus of this thesis.

The story of SAT is one of evolution and interdisciplinary impact, bridging theoretical insights with practical applications. It is a testament to how fundamental questions in logic and mathematics can lead to tools and technologies that shape how we solve problems in science, engineering, and beyond.

1.3 Thesis Statement

This thesis aims to explore new fields of research in Boolean Satisfiability besides classical satisfiability-solving techniques, using modern machine learning (ML) and deep learning (DL) methods. The Boolean Satisfiability Problem and its extension, Propositional Model Counting, are foundational challenges in computational logic, posing significant theoretical and practical problems. These problems are at the core of computational complexity theory, with SAT being the first identified NP-complete problem and #SAT classified as #P-complete. Despite decades of research, the efficient resolution of these problems for increasingly complex instances remains a crucial challenge.

The focus of this thesis is twofold: improving the automation and performance of SAT problems analysis through ML and DL techniques, and advancing the generation of synthetic SAT problem instances for benchmarking purposes. Traditional SAT-solving methods, such as the DPLL and CDCL algorithms, have achieved remarkable success but rely heavily on handcrafted heuristics and rule-based approaches. These methods face limitations in scalability and adaptability to the diverse structures encountered in modern SAT instances, particularly in domains like hardware verification, combinatorial optimization, and AI planning.

This research explores the following fundamental questions:

- Can machine learning models enhance the prediction tasks in SAT solving, such as SAT/UNSAT classification, problem categorization, and model counting?
- What are the benefits of applying deep learning architectures, including Convolutional Neural Networks (CNNs) and Graph Neural Networks (GNNs), to SAT problems?
- How can automatic feature extraction methods, particularly those relying on autoencoders, replace traditional handcrafted approaches to improve speed and generalization?
- What are the most effective methodologies for generating realistic SAT instances using generative models like Graph Variational Autoencoders (GVAEs)?
- How can these generative models be evaluated to ensure the produced instances are structurally and statistically diverse, encompassing real-world tasks, and useful for benchmarking solver performance?

1.4 Contributions

This thesis introduces several significant contributions to the integration of machine learning and deep learning techniques in SAT applications, focusing on improving the efficiency, accuracy, and interpretability of SAT/UNSAT classification, model counting, feature extraction, and SAT instance generation:

- Developed and deployed machine learning models for predicting SAT/UNSAT, problem category classification, and model counts of SAT instances. These

models utilize diverse feature sets derived from SAT instances, combining statistical metrics (e.g., clause-variable ratios) and structural representations (e.g., graph-based features) to capture the intrinsic properties of SAT problems.

- Enhanced the explainability of machine learning models by employing feature importance analysis methods such as SHAP (SHapley Additive exPlanations). This analysis identifies which features most significantly impact predictions for SAT/UNSAT classification, problem categorization, and model counting, providing insights into model behavior and aiding solver optimization.
- Extended the application of deep learning models, including Convolutional Neural Networks, Graph Neural Networks, Convolutional Autoencoders, and Graph Variational Autoencoders, to perform SAT/UNSAT classification and instance category prediction. These models achieved high predictive accuracy by leveraging data-driven approaches, eliminating the need for handcrafted feature engineering. This contribution builds on the earlier machine learning models by offering fully automated and scalable prediction frameworks.
- Automated the feature extraction process for SAT instances using autoencoders, significantly reducing reliance on handcrafted features. Autoencoders learn representations directly from raw SAT data, streamlining the feature engineering process, and enhancing processing efficiency. By replacing manually designed features with data-driven representations, this approach accelerates solver deployment and improves adaptability to diverse SAT instance distributions.
- Conducted a comprehensive comparative analysis of traditional SAT solvers and machine learning/deep learning-based methods. The comparison highlights the strengths and weaknesses of each approach in terms of predictive accuracy, computational efficiency, and interpretability, providing a roadmap for hybrid solver strategies.
- Proposed a novel approach for generating realistic and diverse SAT instances using a Graph Variational Autoencoder model. This model captures the statistical and structural properties of real-world SAT instances and generates synthetic benchmarks suitable for testing and improving SAT solvers.

- Introduced new evaluation metrics for assessing SAT instance generation quality, focusing on diversity, realism, and solver performance on the generated problems. These metrics provide deeper insights into the practical utility of the generated instances and set a foundation for benchmarking future SAT generative models.

Practical Implications of Predictive Models in SAT Solving. The predictive models developed in this thesis, consisting in satisfiability classification, instance categorization, and model counting, offer tangible benefits in enhancing SAT solving efficiency. By accurately predicting whether an instance is satisfiable, one can pre-emptively determine the applicability of SAT or MaxSAT solvers, thus optimizing computational resources [XHLB10]. Instance classification into categories enables the selection of solvers specifically tailored to the structural characteristics of each instance type, as demonstrated by portfolio-based approaches like SATzilla [XHHLB08] and other ML-supported solver selection methods [LPPR19]. Furthermore, insights gained from model counting can inform decisions regarding solver heuristics and instance complexity assessment, leading to more efficient solving strategies overall [ZCC+24]. Although feature extraction introduces computational overhead, studies indicate that strategic application of predictive insights consistently leads to overall reductions in solving time [KMS+11]. In essence, this thesis seeks to extend the knowledge of SAT solving by integrating the strengths of traditional techniques with the adaptability and power of modern ML and DL approaches. The outcomes are expected to have an impact on the efficiency and applicability of SAT solvers, contributing to advancements in academic research and real-world applications.

1.5 Publications

The research contributions of this thesis have resulted in the following publications:

- Dalla, M., Visentin, A., & O'Sullivan, B. (2021). Automated SAT Problem Feature Extraction Using Convolutional Autoencoders. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)* (pp. 232–239). IEEE.
- Provan-Bessell, B., Dalla, M., Visentin, A., & O'Sullivan, B. (2022). SATfeatPy–

A Python-based Feature Extraction System for Satisfiability. *arXiv preprint arXiv:2204.14116*.

- Bergin, R. H., Dalla, M., Visentin, A., O'Sullivan, B., & Provan, G. (2023). Using Machine Learning Classifiers in SAT Branching. In *Proceedings of the International Symposium on Combinatorial Search*, 16(1), 169–170.
- Dalla, M., Provan-Bessell, B., Visentin, A., & O'Sullivan, B. (2023). SAT Feature Analysis for Machine Learning Classification Tasks. In *Proceedings of the International Symposium on Combinatorial Search*, 16(1), 138–142.
- Crowley, D., Dalla, M., O'Sullivan, B., Visentin, A., (2024). SAT Instances Generation Using Graph Variational Autoencoders. In *ESANN 2023 Proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*.
- Dalla, M., Visentin, A., O'Sullivan, B., (2024). A Machine Learning Approach to Model Counting. In *The 36th IEEE International Conference on Tools with Artificial Intelligence*.

1.6 Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2: Background and State of the Art** reviews Boolean Satisfiability, model counting, and modern SAT solving techniques, along with an overview of machine learning and deep learning methods applied to SAT problems.
- **Chapter 3: Machine Learning Predictions Based on SAT Features** describes our approach to feature extraction and presents the machine learning models used for predicting SAT/UNSAT classification, problem category, and model counting.
- **Chapter 4: Deep Learning Predictions** discusses the application of deep learning models to SAT problems, focusing on SAT/UNSAT prediction, problem category, model counting and automatic feature extraction.
- **Chapter 5: SAT Generation with Deep Learning Models** details our approach to generating SAT instances using graph-based variational autoencoder.

- **Chapter 6: Conclusions and Future Work** summarizes the contributions of the thesis and outlines directions for future research.

Chapter 2

Background

2.1 Artificial Intelligence

Artificial Intelligence (AI) is a multidisciplinary field focused on developing systems capable of performing tasks that typically require human intelligence, such as reasoning, learning, problem-solving, perception, and language understanding [RN09]. AI integrates concepts from computer science, mathematics, cognitive science, neuroscience, and philosophy to create algorithms and models that can process complex data and make autonomous decisions.

2.1.1 Definitions and Perspectives

Russell and Norvig [RN09] categorize AI definitions into four approaches:

- **Systems that think like humans:** This approach involves creating machines that emulate human thought processes, often by modelling cognitive functions.
- **Systems that act like humans:** Here, the focus is on producing behaviour indistinguishable from human behaviour, as exemplified by the Turing Test [Tur50].
- **Systems that think rationally:** This perspective aims to formalize reasoning processes using logical systems to produce rational thought.
- **Systems that act rationally:** Emphasizes intelligent behaviour through rational actions to achieve specified goals, often modelled using agents.

Despite differing perspectives, a unifying theme is the development of computational systems that can solve complex problems using sophisticated techniques.

2.1.2 Current Developments in AI

Recent advancements have led to significant breakthroughs in various AI domains, which we briefly summarise below.

Natural Language Processing (NLP). The introduction of transformer architectures [VSP+17] has revolutionized NLP by enabling models to capture long-range dependencies in text. Models like BERT [DCLT18] and GPT-3 [B+20] achieve state-of-the-art results in tasks such as language translation, sentiment analysis, and question answering. These models pre-train on vast amounts of textual data, learning contextual relationships that improve language understanding and generation.

Computer Vision. Deep learning, particularly convolutional neural networks (CNNs), has dramatically improved image classification, object detection, and segmentation tasks [HZRS16]. Architectures like ResNet [HZRS16] and EfficientNet [TL19] have pushed the boundaries of performance on benchmark datasets like ImageNet. These advancements have enabled widespread applications, from autonomous vehicles that interpret complex visual scenes to medical diagnostics that assist in detecting diseases from imaging data.

Reinforcement Learning and Decision Making. Reinforcement learning (RL) algorithms have excelled in complex decision-making tasks by learning optimal policies through trial-and-error interactions with the environment. Notably, AlphaGo [S+16] and its successors AlphaZero and MuZero have defeated human champions in games like Go, Chess, and Shogi, demonstrating the potential for AI in strategic planning and problem-solving. These achievements are based on combining RL with deep neural networks and advanced search techniques.

Generative Models. Generative models, such as Generative Adversarial Networks (GANs) [GPAM+14] and Variational Autoencoders (VAEs) [KW13], have advanced the ability of AI systems to generate realistic data, including images, audio, and text. These models have applications in data augmentation, creative content generation, and simulation, contributing to advancements in fields like art, music, and virtual reality.

2.1.3 Challenges and Ethical Considerations

Despite progress, AI development faces significant challenges:

Data Quality and Bias. AI models depend on large datasets, which may contain biases leading to unfair or discriminatory outcomes [BHN19]. For example, facial recognition systems have exhibited higher error rates for certain demographic groups due to imbalanced training data [BG18]. Addressing data quality and implementing fairness measures are essential for ethical AI deployment, requiring techniques for bias detection and mitigation.

Explainability and Transparency. Complex models, especially deep neural networks, often act as "black boxes," lacking interpretability [Lip18]. This opacity makes it difficult to understand their decision-making processes, which is problematic in high-stakes domains like healthcare and finance. Explainable AI (XAI) aims to make AI systems more transparent to enhance trust, facilitate debugging, and ensure compliance with regulatory requirements [A$^+$20].

Ethical Governance. The ethical implications of AI have prompted regulatory efforts worldwide. The European Union's proposed AI Act [Com21] seeks to establish a legal framework for trustworthy AI, promoting innovation while safeguarding fundamental rights. The Act categorizes AI applications based on risk levels and imposes obligations on providers, including transparency, accountability, and human oversight.

Robustness and Safety. Ensuring AI systems are robust against adversarial attacks and can handle unexpected inputs is critical, especially in safety-critical applications like healthcare and autonomous driving [CW17]. Research in adversarial machine learning explores how malicious inputs can fool AI models and develop defenses to enhance system resilience [AM18].

Privacy Concerns. AI systems often process sensitive personal data, raising privacy issues. Techniques like federated learning [MMR$^+$17] and differential privacy [Dwo08] are being developed to enable model training without compromising individual privacy.

2.1.4 Algorithms in AI

AI employs a variety of algorithms to solve diverse problems. While many foundational algorithms exist, this section highlights those most relevant to machine learning (ML) and its applications.

Search Algorithms. Search algorithms, such as A* and Dijkstra's, are essential for navigating large problem spaces efficiently, often used in pathfinding and game playing [HNR68].

Optimization Algorithms. Optimization algorithms like Genetic Algorithms (GA) and Simulated Annealing (SA) are used to find optimal solutions in complex, multidimensional spaces where an exhaustive search is impractical [Hol92].

Probabilistic Algorithms. Probabilistic algorithms, such as Bayesian Networks, are key in reasoning under uncertainty and are widely used in fields like diagnostic systems and decision-making [Pea88].

Machine Learning Algorithms. Machine learning is a core subset of AI focused on developing algorithms that enable computers to learn from data and make decisions. ML models are distinct in that they improve their performance over time with minimal human intervention. This is achieved through various learning paradigms, including supervised, unsupervised, and reinforcement learning [GBC16]. A more in-depth description of machine learning algorithms will be discussed in Section 2.3.

2.2 Boolean Satisfiability

The Boolean satisfiability problem [Coo71], denoted as SAT, asks whether a given propositional logic formula Φ has a variable assignment $\mathcal{V}$ that satisfies it, i.e., makes the formula evaluate to true. A propositional formula, also known as a Boolean expression, is constructed from variables, constants (true or false), negations ($\neg$), and connectives such as AND ($\wedge$), OR ($\vee$), IMPLIES ($\rightarrow$), or EQUIVALENT ($\leftrightarrow$) [DP60].

A variable assignment $\mathcal{V}$ assigns a truth value (true or false) to each variable in Φ. If $\mathcal{V}$ makes Φ evaluate to true, denoted as $\mathcal{V}(\Phi) = \text{true}$, then $\mathcal{V}$ is called a *model* of Φ. The formula Φ is satisfiable if there exists at least one model $\mathcal{V}$

such that $\mathcal{V} \models \Phi$. If every possible variable assignment is a model of Φ, then Φ is a tautology, denoted as $\models \Phi$. Conversely, if no variable assignment satisfies Φ, the formula is unsatisfiable, and its negation is a tautology.

For example, consider the formula $\Phi = (x \vee y) \wedge (\neg x \vee \neg y)$. A variable assignment that satisfies Φ is $\mathcal{V}(x) = \text{true}$ and $\mathcal{V}(y) = \text{false}$. Under this assignment, both clauses in Φ evaluate to true, so $\mathcal{V}$ is a model of Φ.

SAT solvers not only determine the existence of a model but often find a specific one if the formula is satisfiable. Advanced SAT solvers can also produce an unsatisfiability proof or identify an unsatisfiable core, which is a minimal subset of clauses that cannot be satisfied.

2.2.1 Model Counting

Model counting, also known as #SAT, extends the SAT problem by asking for the (exact) number of distinct models that satisfy a given formula Φ [Tod89]. While SAT is a decision problem concerned with the existence of at least one model, model counting is a counting problem that quantifies the total number of satisfying variable assignments. Determining this number is generally a more computationally challenging task because it requires accounting for all possible combinations of variable assignments that satisfy the formula rather than just finding one [DJW05].

In computational complexity theory, SAT is classified as an NP-complete problem. This means that while any proposed solution (a satisfying assignment) can be verified quickly (in polynomial time), there is no known algorithm that can find a solution quickly for all instances. Model counting, on the other hand, is classified as a #P-complete problem [Coo71], which is a complexity class for counting problems associated with the solutions of NP problems. #P-complete problems are considered even more computationally intractable because they involve enumerating all possible solutions, and there is no known polynomial-time algorithm to solve them.

Continuing with the previous formula $\Phi = (x \vee y) \wedge (\neg x \vee \neg y)$, there are two satisfying assignments:

1. $\mathcal{V}_1(x) = \text{true}, \quad \mathcal{V}_1(y) = \text{false}$ 2. $\mathcal{V}_2(x) = \text{false}, \quad \mathcal{V}_2(y) = \text{true}$

Therefore, the model count for Φ is 2.

2.2.2 Conjunctive Normal Form

While the syntax of propositional logic is tailored for manual formula manipulation, a more streamlined structure is beneficial for algorithmic handling. The structure of choice for SAT solvers is the conjunctive normal form (CNF) [DP60].

The significance of the conjunctive normal form for SAT solving was first emphasized by Davis and Putnam. A CNF formula is characterized by a conjunction of clauses. Each clause $\mathcal{C}$ is a disjunction of literals, where a literal is either a variable or its negation. This structure permits a straightforward representation as a collection of sets of literals.

$$Formula ::= Clause \wedge Formula$$

$$Clause ::= Literal \vee Clause$$

$$Literal ::= Variable \vee \neg Variable$$

$$Variable ::= A \vee B \vee C \vee \dots$$

The grammatical notation used to define the syntax of propositional formulas follows the Backus-Naur Form (BNF) [MR03]. Any propositional formula can be transformed into an equivalent CNF formula using basic rewrite rules. However, this transformation can cause some formulas to expand exponentially in size. For instance, the formula:

$$(A_1 \wedge B_1) \vee (A_2 \wedge B_2) \vee \cdots \vee (A_n \wedge B_n)$$

has a corresponding CNF with 2^n clauses, where n is the number of conjunctive terms in the disjunction, i.e. the number of original clauses. This exponential growth occurs because each conjunctive term must be distributed over all others, leading to all possible combinations of literals.

For SAT solving, it is not mandatory to use an equivalent formula. Equisatisfiable formulas, which might introduce new variables, are sufficient [BHvMW09]. For CNF conversion, these new variables typically represent the value of a subformula, allowing for a conversion where the output length is polynomially related to the input length.

2.2.3 The DPLL Procedure

There are several methods in the literature that form the basis for most SAT solver implementations. The Davis-Putnam [DP60] procedure, often referred to as DPP, was a pioneering satisfiability testing method and can be seen as an antecedent to contemporary SAT solvers. It introduced the CNF into satisfiability testing. DPP comprises three rules that modify a set of CNF clauses without altering its satisfiability. Through repetitive application, the set is either reduced to the empty set (indicating satisfiability) or to a set containing an empty clause (indicating unsatisfiability). The rules are:

- **Unit-Clause Rule:** If a clause consists of a single literal, that literal must be true for the formula to be satisfied. By assuming this literal's value as true, all clauses containing that literal are satisfied and can be discarded. Clauses containing the negated literal are modified by removing the negated literal.
- **Pure-Literal Rule:** If a literal appears but its negation does not, it is safe to assume the literal's value as true. This cannot make any clause unsatisfiable. All clauses containing this literal can be removed. While historically significant, the pure literal rule has largely been abandoned in modern complete SAT solvers, as its practical benefit has proven minimal in contemporary solver implementations.
- **Resolution Rule:** This rule aims to eliminate a variable from the set of clauses. All clauses containing the variable are grouped based on positive or negative occurrence. These clauses are then replaced with the disjunction of both sets. When converted back to CNF, it becomes the set of all disjunctions formed by taking a clause from either set.

However, the resolution rule can quickly generate a vast number of clauses, leading to memory constraints. This challenge led Loveland and Logemann to replace it with a splitting rule [ES04] that successively explores the implications of assuming a variable to be true or false, resulting in a recursive algorithm. This became the DPLL procedure, which remains foundational for most modern SAT solvers.

The DPLL procedure rules are further explained in the following examples:

- **Unit-Clause Example:** Given the clauses $\{A, B, \neg C\}$, $\{\neg B, C\}$, and $\{\neg B, \neg C\}$, applying the unit clause rule with $B = \text{true}$ results in the clauses $\{A\}$ and

$\{\neg C\}$.

- **Pure-Literal Example:** Given the clauses $\{A, B, C\}$, $\{\neg A, B\}$, and $\{\neg A, C\}$, applying the pure-literal rule with $B = \text{true}$ (since $\neg B$ does not appear in any clause) allows us to remove all clauses containing B. This results in the remaining clause $\{\neg A, C\}$.
- **Resolution Example:** Given the clauses $\{A, B\}$ and $\{\neg B, C\}$, applying the resolution rule on variable B results in the clause $\{A, C\}$. Here, we resolve B and $\neg B$ by combining the remaining literals after eliminating B and $\neg B$.
- **Splitting the clause:** Splitting the clause $(A \vee B \vee C \vee D)$ into two clauses $(A \vee B \vee X) \wedge (C \vee D \vee \neg X)$ where X is a fresh variable is an example of a transform that preserves satisfiability but not equivalence.

with A,B,C,D,X are variables.

The pseudocode for the DPLL procedure is as follows:

Algorithm 1 DPLL Procedure

Require: Φ {The formula in CNF}
Ensure: Satisfiability status of Φ

```
Procedure DPLL(Φ):
Φ ← Propagate(Φ)
if ∅ ∈ Φ then
    return unsat
else if Φ = ∅ then
    return sat
end if
Choose v from Vars(Φ)
if DPLL(Φ[v = true]) = sat then
    return sat
end if
if DPLL(Φ[v = false]) = sat then
    return sat
end if
return unsat
```

2.2.4 Lookahead

The DPLL algorithm does not provide explicit guidance on the selection of variables for branching. The choice of the decision variable can significantly influence the algorithm's overall efficiency. Lookahead solvers [HvM09] aim to

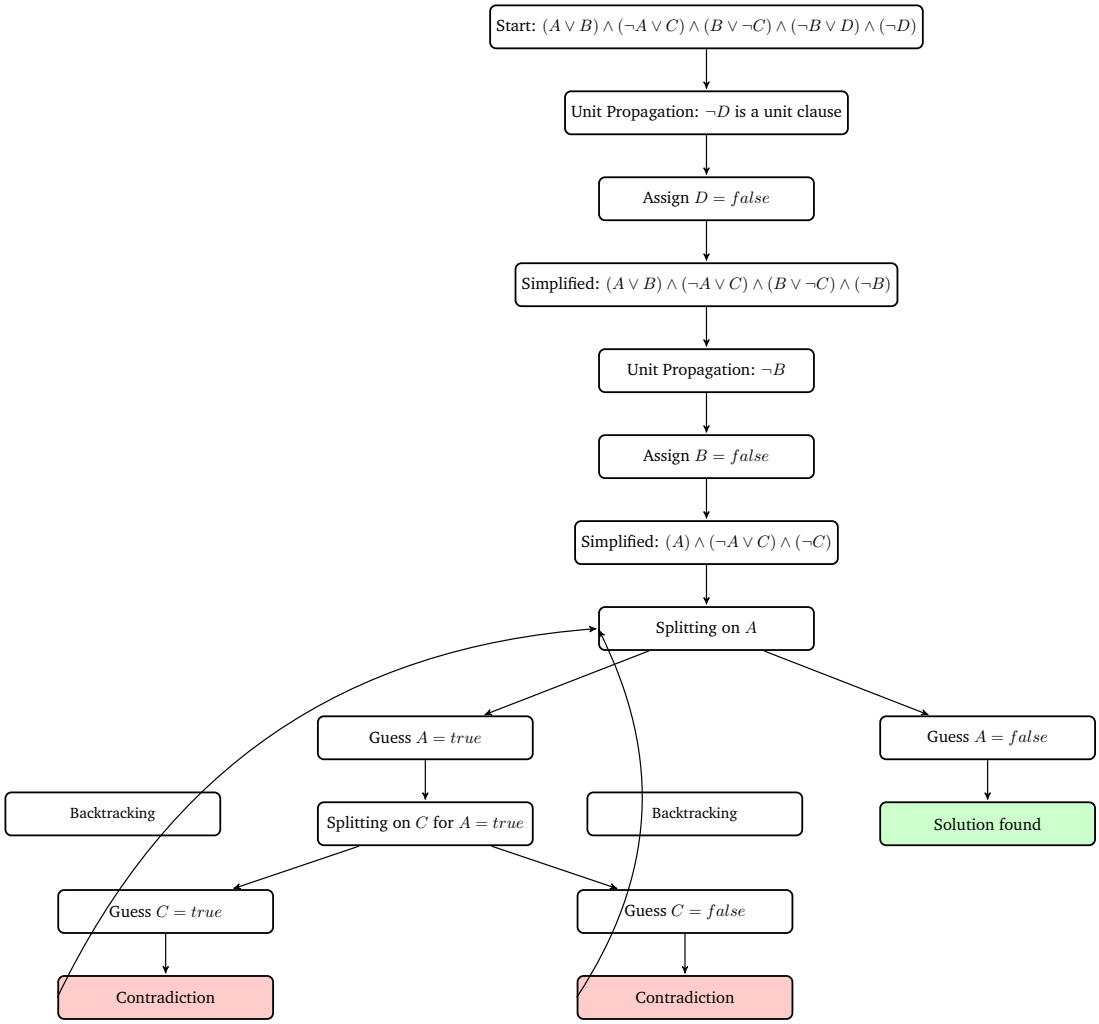


Figure 2.1: Execution of the DPLL procedure.

make informed decisions by examining the potential outcomes of branching on specific variables. This is achieved through a lookahead procedure that triggers the propagate procedure for both possible assignments of a variable (see Line 3 in algorithm 2).

Variables are subsequently ranked based on a score derived from the resulting CNF formulas (Lines 12–15 in 2). If a conflict arises when assuming a literal during the lookahead process—a scenario known as a *failed literal*—the assumed variable must adopt the negated value [MSS99].

In Algorithm 2, after the lookahead procedure (Line 3), the decision variable x is chosen based on the ranking, and the *direction heuristic* is applied to determine the assignment order (Line 6). This heuristic decides whether to try $x =$ true or $x =$ false first, based on criteria such as the one that leads to fewer conflicts during the lookahead.

Algorithm 2 DPLL and Lookahead Procedures

Require: Φ {The formula in CNF}
Ensure: Satisfiability status of Φ
1: **Procedure** DPLL(Φ):
2: $\Phi \leftarrow$ Propagate(Φ)
3: $\Phi, x \leftarrow$ Lookahead(Φ) {Line 3: Invoke lookahead procedure}
4: **if** $\Phi = \emptyset$ **then**
5: **return** sat
6: **else if** $\emptyset \in \Phi$ **then**
7: **return** unsat
8: **end if**
9: $b \leftarrow$ Direction Heuristic(Φ, x) {Line 6: Apply direction heuristic}
10: result$_b \leftarrow$ DPLL($\Phi[x = b]$)
11: **if** result$_b$ = sat **then**
12: **return** sat
13: **end if**
14: result$_{\neg b} \leftarrow$ DPLL($\Phi[x = \neg b]$)
15: **return** result$_{\neg b}$
16: **Procedure** Lookahead(Φ):
17: Choose a set of variables X
18: **for** each x in X **do**
19: Propagate with x = true and x = false
20: **if** Conflict occurs **then**
21: Mark x as a failed literal
22: **end if**
23: **end for**
24: Rank variables based on the results {Lines 12–15: Rank variables}
25: Choose x with the highest rank
26: **return** Φ, x

2.2.5 Conflict-Driven Clause Learning

An enhancement to the DPLL procedure involves the dynamic incorporation of new clauses into the formula, aiming to steer the search towards a solution. This enhancement is achieved by examining conflicts that arise during the recursive process and subsequently introducing clauses that prevent similar conflicts in subsequent iterations. This strategy is termed conflict-driven clause learning (CDCL) [MSLM09].

The identification of a clause that can prevent a previously observed conflict leverages an implication graph [ZMMM]. This graph is a directed representation of all literals deemed true based on the current set of assumptions. Each time the unit-clause rule is invoked, new directed edges are established, point-

ing towards the unit clause's literal from every negated literal of the original clause that was identified as a unit.

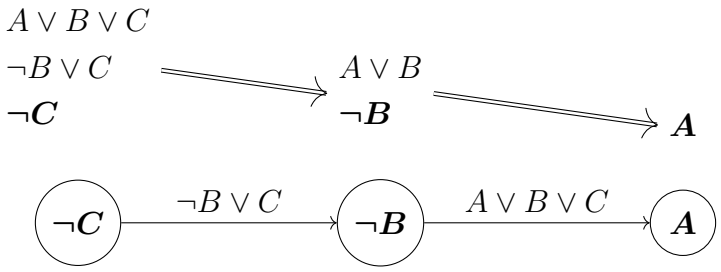


Figure 2.2: Example of an implication graph

A conflict is identified when every literal negation from a clause is found within the implication graph in Figure 2.2. To derive a clause that can avert this conflict in subsequent iterations, a graph cut is determined between the literals assumed and those causing the conflict. A clause encompassing all negated literals from nodes with outgoing edges that intersect the cut is then appended to the formula.

Consider the implication graph shown in Figure 2.2. Assume the SAT solver identifies a conflict involving the literals A, $\neg B$, and $\neg C$. These literals imply a contradiction because the clause $\neg C \vee B \vee A$ becomes unsatisfiable.

To resolve this, a graph cut is made to separate the assumed literals ($\neg C$, $\neg B$) from the conflict node. The literals with outgoing edges that cross the cut ($\neg C$ and $\neg B$) are collected, and a new clause $C \vee B$ is derived. Adding this clause to the formula ensures that the same conflict does not recur under the same assumptions.

The selection of the graph cut is determined heuristically [ZMMM]. Furthermore, the implication graph can be implicitly represented by noting down the clause responsible for implying a specific literal.

Upon adding such a clause to the formula, in the absence of backtracking, all its literals are rendered false. Based on the chosen graph cut, it might be essential to backtrack across multiple levels until the newly added clause becomes a unit. This clause can be a unit across various levels, and empirical evidence suggests that backtracking to the most inferior level where the clause is a unit, a technique known as non-chronological backtracking or backjumping, is efficacious.

Given that backtracking can span multiple levels concurrently, CDCL is typically implemented in an iterative manner, accompanied by a clear stack of assumed

literals. This approach is depicted in the structure showcased in the pseudocode of Algorithm 3. The pioneering SAT solver that ushered in this methodology, in tandem with non-chronological backtracking, was named GRASP [MSS99]. Another solver worth noting is Chaff, credited with the introduction of a highly efficient data structure for unit-propagation termed as watched literals [MM$^+$01]. A comprehensive guide detailing the evolution of a CDCL solver is available in [ES04], which elaborates on the solver minisat.

Algorithm 3 CDCL Algorithm

Require: Φ {The formula in CNF}
Ensure: Satisfiability status of Φ
1: $level \leftarrow 0$
2: **while** true **do**
3: **if** ϕ[assumptions] $= \emptyset$ **then**
4: **return** sat
5: **else if** $\emptyset \in \phi$[assumptions] **then**
6: **if** $level = 0$ **then**
7: **return** unsat
8: **end if**
9: $c \leftarrow$ AnalyzeConflict(assumptions, reasons)
10: $\phi \leftarrow \phi \cup \{c\}$
11: backtrack to smallest $level$ so that $|c$[assumptions]$| = 1$
12: **else if** $c \in \phi$ with c[assumptions] $= \{l\}$ **then**
13: assumptions$_l evel \leftarrow$ assumptions$_l evel \cup \{l\}$
14: reasons$_l \leftarrow c$
15: **else**
16: $level \leftarrow level + 1$
17: choose unassigned literal l
18: assumptions$_l evel \leftarrow$ assumptions$_l evel \cup \{l\}$
19: **end if**
20: **end while**

2.2.5.1 Conflict Analysis and Clause Resolution

Modern CDCL SAT solvers predominantly utilize the first Unique Implication Point (fUIP) [MSS99] technique for clause learning. Beginning with the conflicting clause $\mathcal{C}$, the fUIP method resolves literals from the present decision level until a clause $\mathcal{L} = \mathcal{R} \cup \{\neg A\}$ is identified. Here, the literal A was determined in the current decision level, while all literals in $\mathcal{R}$ were determined previously. The literal F is termed the fUIP literal for the ongoing conflict. Every path from the current decision variable to the ongoing conflict contains the fUIP literal F. The literals in $\{r|\text{literal } r \in \mathcal{R}\} \cup \{A\}$ are termed reason literals

for the ongoing conflict, as their assignments resulted in the current conflict. $\mathcal{R}$ is termed the reason clause for the ongoing conflict with conflicting clause $\mathcal{C}$.

2.2.5.2 Fast Conflict Creation

During the search process, when a conflict arises, a CDCL SAT solver learns a clause from that conflict. These learned clauses play a pivotal role in trimming the search space, thereby enhancing the efficiency of solving. Research has shown [BKS04] that for UNSAT formulas, the shortest refutation proofs identified by the CDCL SAT solving process with clause learning are only marginally longer than the shortest refutation proofs generated by general resolution, a robust proof system. Efficient branching heuristics have been observed to produce more conflicts per branching decision and derive higher-quality clauses from these conflicts [Rya04]. Thus, swiftly generating conflicts and deriving high-quality clauses from these conflicts is crucial for effective CDCL SAT solving.

2.2.5.3 CDCL Branching Techniques

The conventional CDCL SAT branching techniques adhere to the look-back principle [MM$^+$01]. They give preference to variables that have recently been part of conflicts, with the belief that these variables, if reassigned, will continue to produce more conflicts. Two notable CDCL branching techniques, VSIDS and LRB, are discussed below.

VSIDS. The Variable State Independent Decaying Sum (VSIDS), proposed by [MM$^+$01], is a widely adopted dynamic branching technique. VSIDS assigns an activity score to each variable in a given SAT formula. During conflict analysis, VSIDS augments the activity score of each variable involved in conflict resolution. It places a significant emphasis on variables that were part of the latest conflicts.

LRB. In the Learning Rate Branching (LRB) [LOM$^+$18] technique, a variable V is considered active when it is assigned due to a branching decision or propagation and becomes inactive when it is unassigned due to backtracking. LRB monitors the participation count $P(V)$ of V in generating learned clauses within the conflict interval $I(V)$. An activity score for V is derived from its reward $R(V)$, which is the rate of its participation in learned clause generation. During

the search, the variable with the highest activity score is chosen for branching.

2.2.5.4 CDCL Polarity Heuristics

After a variable is decided by the decision techniques, a CDCL SAT solver employs a polarity technique to assign a boolean value to the decided variable. The most advanced polarity technique is the phase-saving technique [ES04].

The Phase Saving Technique. Consider two literals, A^+ and A^-, where the variable A is assigned true and false, respectively. Assume that at decision step i, a variable B is assigned, leading to a propagation with A^x, where $x \in \{+, -\}$, and this propagation results in a conflict. After analyzing the conflict and during backtracking, the phase-saving technique retains this last assigned polarity x of A in polarity[A]. If the CDCL decision technique selects A again at decision step $j > i$, the phase-saving technique selects the literal A^x. In CDCL SAT solvers, such as Glucose and Maple [PD07], the polarity[x] is initialized to $-$ each variable A of a given SAT formula. This initialization to negative polarity is an artifact of the encoding of most SAT benchmarks, which generally produce formulas with more positive than negative literals.

2.2.5.5 Metrics Related to Conflicts

This section reviews three pertinent conflict metrics: Global Learning Rate (GLR), Literal Block Distance (LBD) Score, and Glue Clauses.

Global Learning Rate. The Global Learning Rate (GLR) [AS12] measures the efficiency of a solver in producing conflicts and learning from them. It is defined as the ratio of the number of conflicts to the number of decisions. A higher GLR indicates that the solver is encountering more conflicts per decision, which typically results in the rapid generation of learned clauses, enabling a more efficient traversal of the search space.

Literal Block Distance Score. The Literal Block Distance (LBD) score [AS09] of a learned clause $\mathcal{C}$ represents the number of distinct decision levels present in $\mathcal{C}$. Lower LBD scores indicate that the variables within the clause are closely interlinked, making these clauses more reusable in future propagations. Clauses with a low LBD score are considered "high quality" because they are more effective in pruning the search space, which contributes to the solver's overall

efficiency.

Glue Clauses. Glue clauses [AS09] are a subset of learned clauses characterized by an LBD score of 2. These clauses are particularly impactful as they connect a literal from the current decision level with a block of literals determined in a previous decision level. Glue clauses are highly effective at reducing the search space and are often prioritized by solvers for retention in the clause database. By focusing on such high-impact clauses, solvers can maintain a lean database while enhancing propagation speed and decision quality.

2.2.5.6 Leading CDCL Solver Systems

In recent times, three systematic solver families have dominated the SAT competitions [FHI+21]:

- **The Glucose Family:** Glucose and its extensions utilize VSIDS and its variants as their branching heuristic. They also employ rapid restart strategies and aggressive clause database cleaning schedules [AS09].
- **The Maple Family:** MapleCOMSPS and its variants use a combination of branching heuristics like VSIDS, LRB, and Dist. These solvers are characterized by less aggressive clause database reduction and restart policies compared to the Glucose family [LKP+17].
- **The CaDiCaL and Kissat Family:** Both CaDiCaL and its successor Kissat are high-performance CDCL SAT solvers. They interleave search and inprocessing, a technique that simplifies the current clause collection. Both solvers use VSIDS and Variable Move To Front (VMTF) as their branching heuristics and are known for their ultra-rapid restarts during the search [FH20].

2.2.6 Local Search Solvers in SAT

Local search solvers are another important category of algorithms employed in SAT problems [SKC94]. These solvers are especially effective for addressing hard combinatorial problems where traditional complete methods, such as the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, may struggle due to the exponential size of the search space. The key strength of local search lies in its ability to quickly find a solution by exploring a vast number of potential solutions without the need to exhaustively search the entire space.

2.2.6.1 Basic Concepts

Local search algorithms are built upon a few fundamental concepts that guide the search process:

Neighborhood. The neighborhood of a given solution refers to the set of all possible solutions that can be obtained by making small, localized changes to the current solution. In the context of SAT, this typically means flipping the value of one or more variables in a given assignment. The choice of neighborhood structure is critical, as it affects the efficiency of the search. Smaller neighborhoods allow for finer-grained exploration, while larger neighborhoods can enable the algorithm to escape local optima more effectively [SKC94, HS04].

Objective Function. In SAT, the objective function often measures the number of satisfied clauses in the current assignment. The goal of the local search algorithm is to maximize this objective function, ideally reaching a point where all clauses are satisfied, which corresponds to a solution of the SAT problem. Different strategies can be employed to evaluate the objective function, such as focusing on hard-to-satisfy clauses or giving more weight to certain constraints based on their complexity or importance [HS04, LH05].

Fitness Landscape. The fitness landscape is a conceptual model that represents how the quality of solutions (as measured by the objective function) varies across the search space. A rugged landscape with many peaks and valleys represents a challenging optimization problem, as the algorithm can easily become trapped in local optima. Understanding the structure of the fitness landscape can help in designing more effective local search strategies [HS04].

Termination Criteria. The search process must have a well-defined stopping condition. Common termination criteria include reaching a maximum number of iterations, exceeding a time limit, or finding a solution that satisfies all constraints. In some cases, the algorithm may also terminate when it detects that it is unlikely to find a better solution, such as when the improvements in the objective function become negligible over successive iterations [GW93, SLM92].

2.2.6.2 Main Algorithms

Several well-known algorithms embody the principles of local search and have been tailored to the SAT problem:

Hill Climbing. Hill climbing is one of the simplest and most intuitive local search methods. It begins with a random solution and iteratively makes small changes, each time moving to a neighboring solution that improves the objective function. Although straightforward, hill climbing is prone to getting stuck in local optima, especially in complex landscapes typical of SAT problems. Variants such as stochastic hill climbing introduce randomness in the selection process to mitigate this issue [GW93].

Simulated Annealing and Tabu Search. Historically, optimization methods such as Simulated Annealing and Tabu Search have been explored for SAT solving, aiming to enhance hill climbing algorithms by avoiding local optima through stochastic decisions or memory-based mechanisms [KGV83, AS95]. However, in modern competitive SAT solving practice, these methods are rarely employed, as stochastic local search algorithms like WalkSAT and its variants have proven significantly more effective and thus dominate current SAT solver competitions.

WalkSAT and Variants. WalkSAT is a specific local search algorithm designed for SAT problems. It starts with a random assignment and iteratively flips the value of a variable to increase the number of satisfied clauses. If no improvement is possible, the algorithm may flip a variable randomly, introducing a level of stochasticity that helps escape local optima. Variants of WalkSAT incorporate additional heuristics, such as clause weighting and configuration checking, to improve performance on harder SAT instances [SKC94, CS12].

Hybrid Approaches. Hybrid algorithms combine local search with other techniques, such as clause learning from Conflict-Driven Clause Learning (CDCL) solvers or lookahead strategies. These approaches leverage the strengths of different algorithms to improve overall performance. For instance, local search might be used to quickly find a good starting point, after which a complete solver refines the solution [LH05, HKWB11].

2.2.6.3 Application of Local Search to SAT

Local search methods have been widely applied in SAT solving, particularly for large, randomly generated instances where complete methods are less practical. The success of these algorithms in SAT is due to their ability to explore large search spaces quickly and their flexibility in handling different problem structures. Additionally, modern SAT solvers often integrate local search components to enhance their performance, especially in hybrid systems.

One significant area of research is the development of specialized heuristics that improve the efficiency of local search in SAT. For example, configuration checking heuristics help prevent cycling by ensuring that recently flipped variables are not flipped again unless necessary, which can greatly enhance the effectiveness of local search on complex instances [CS12].

2.2.7 State of the Art in SAT Solving

The field of SAT solvers has evolved significantly over the past decades, with Conflict-Driven Clause Learning (CDCL) solvers at the forefront of this progress [MSS99]. These solvers have been instrumental in tackling complex combinatorial problems, leading to breakthroughs in areas such as hardware verification, artificial intelligence, and cryptographic analysis. This section reviews the latest advancements in SAT solving, highlighting key improvements in CDCL algorithms, clause management, restart strategies, and the integration of machine learning techniques. The fusion of traditional heuristics with data-driven learning approaches marks a new era in SAT solver efficiency, scalability, and applicability, as evidenced by recent literature and benchmark studies [BHvMW09, LOM+18, Wu17].

2.2.7.1 Improvements in Conflict-Driven Clause Learning

Recent developments in CDCL algorithms have introduced a variety of enhancements aimed at optimizing the performance and scalability of SAT solvers. These advancements can be summarized in the following key areas:

Enhanced Heuristics for Variable Selection: Modern CDCL solvers have benefited from more sophisticated branching heuristics. Techniques such as the Variable State Independent Decaying Sum (VSIDS) have been further refined, leading to more efficient decision-making processes during the search. Recent

studies have proposed adaptive heuristics that dynamically adjust based on the problem structure, significantly reducing solver runtimes [LOG+18, BS22].

Clause Database Management: Effective management of the growing clause database remains crucial for maintaining solver efficiency. Innovations in clause deletion policies, such as those based on clause activity and the LBD (Literal Block Distance) metric, have helped in eliminating redundant or less useful clauses. This has resulted in more streamlined and faster solvers, as highlighted in recent competitions and benchmark studies [AS09].

Learning from Conflicts: The ability of CDCL solvers to learn from conflicts has been significantly enhanced through the incorporation of techniques like XOR-Clause Learning. These techniques allow solvers to manage complex constraints and provide shorter unsatisfiability proofs, thereby improving solver performance on challenging instances [LJN12].

Parallelization and Distributed Solving: With the increasing demand for solving large-scale SAT instances, parallel CDCL solvers have gained traction. Innovations in distributed clause learning and shared clause management have improved the scalability of CDCL solvers on multi-core and distributed systems, demonstrating substantial performance gains in large benchmarks [HJS09, MML12].

These advancements collectively represent a significant step forward in SAT solver technology, ensuring that CDCL remains at the cutting edge of combinatorial problem solving. The continuous refinement of these techniques promises to further expand the applicability of SAT solvers in fields such as hardware verification, AI planning, and cryptographic analysis.

2.2.7.2 Clause Management and Simplification

The continuous addition of clauses by CDCL solvers has necessitated the development of efficient clause management strategies to maintain solver performance. As the clause database expands during the solving process, redundant or irrelevant clauses can accumulate, potentially leading to inefficiencies. Therefore, effective clause management is crucial to reduce the search space without sacrificing the solver's completeness.

Clause Deletion Policies. One of the primary strategies for managing the clause database involves the application of dynamic clause deletion policies. Techniques such as the Literal Block Distance (LBD) metric have been instrumental in identifying and retaining only the most relevant clauses. Clauses with low LBD scores, which tend to be more useful in future conflict analysis, are kept, while others are periodically deleted [AS09]. These policies have been further refined by incorporating adaptive mechanisms that adjust deletion thresholds based on the solver's progress [KLW22].

Clause Activity Measurement. In addition to deletion policies, clause activity measurement has emerged as a crucial factor in clause management. This technique tracks the frequency with which a clause participates in conflict resolution, allowing the solver to prioritize clauses that are more likely to contribute to solving the problem [MSLM09].

Clause Simplification Techniques: Beyond deletion, simplification techniques such as subsumption and self-subsuming resolution have been employed to reduce the size of clauses or eliminate redundant literals. These methods help in streamlining the clause database, thus reducing memory overhead and improving overall solver efficiency [EB05]. Recent work has focused on extending these techniques to handle more complex clauses, including XOR and cardinality constraints, which are common in industrial applications [MHB13].

Impact of Clause Management on Solver Performance: The effectiveness of these clause management techniques is evident in their impact on solver performance. Studies have shown that solvers employing advanced clause deletion and simplification methods consistently outperform those that do not, particularly on large and complex problem instances [HJ12, MHB13]. These improvements are reflected in the results of SAT competitions, where solvers with superior clause management strategies often achieve top rankings.

Overall, the continuous refinement of clause management and simplification techniques plays a critical role in maintaining the efficiency and scalability of modern CDCL solvers. As the complexity of SAT instances grows, further innovations in this area will be essential to keep pace with the increasing demands of real-world applications.

2.2.7.3 Restart Strategies and Incremental Solving

Restart Strategies. Restarts have become a crucial technique in SAT solving to overcome the limitations of traditional backtracking methods. Traditional backtracking can sometimes trap the solver in a local minima, where it explores only a small portion of the solution space, leading to inefficient solving processes. To mitigate this, restart strategies periodically reset the solver's state, enabling it to explore different regions of the solution space. This approach was notably advanced by Huang [Hua07], who demonstrated that systematic restarts can dramatically improve the solver's performance on challenging instances. Since then, various adaptive restart policies have been proposed, where the decision to restart is based on dynamic metrics such as conflict frequency or learned clause quality [Bie08, LOG+18]. These strategies have been shown to reduce the time to solution significantly, particularly in hard SAT instances where solvers might otherwise struggle.

Incremental Solving. Incremental SAT solving represents another key advancement, particularly valuable in applications like hardware verification and automated testing. Incremental solvers reuse information from previously solved instances, which allows them to solve new, but related, instances more efficiently. The FastPass framework, for instance, utilizes incremental solving to optimize pin access schemes in design rule checking, significantly reducing computational overhead [WLY23]. Recent developments have focused on enhancing the robustness of incremental solvers by improving their ability to manage and reuse learned clauses across different but related problem instances [NR12]. This approach not only reduces solving time but also enhances the scalability of SAT solvers in real-world applications.

2.3 Machine Learning

Machine Learning (ML) can be broadly categorized into three primary types, each serving different purposes and employing different algorithms.

2.3.1 Supervised Learning

Supervised learning involves training a model on labeled data, where the correct output is provided for each input [HTF09]. The goal is to learn a mapping

from inputs to outputs that can be generalized to new, unseen data. Common algorithms in supervised learning include the following.

Support Vector Machines (SVM). SVMs are powerful models for classification and regression tasks. They work by finding the hyperplane that best separates data into different classes with the maximum margin, making them effective in high-dimensional spaces [CV95].

Decision Trees. These models split the data into subsets based on the value of input features, forming a tree-like structure that is easy to interpret and implement [Qui86].

Multi-Layer Perceptrons (MLPs). MLPs are a class of feedforward artificial neural networks consisting of an input layer, one or more hidden layers, and an output layer. Each neuron in one layer is fully connected to every neuron in the next layer. MLPs use non-linear activation functions like ReLU or sigmoid to model complex, non-linear relationships in data. They are widely used for supervised learning tasks such as classification and regression due to their ability to approximate any continuous function [RHW86].

2.3.2 Unsupervised Learning

Unsupervised learning deals with unlabeled data, aiming to identify hidden structures or patterns without explicit instruction [Bis06]. The model learns to group similar data points together or reduce the dimensionality of the data. Common techniques include the following.

Clustering Algorithms. Algorithms like K-means are used to partition data into clusters based on similarity, with each data point assigned to the nearest cluster centroid [Bis06].

Dimensionality Reduction. Techniques like Principal Component Analysis (PCA) reduce the number of input variables in a dataset, simplifying the data while preserving as much variance as possible [Jol02].

2.3.3 Reinforcement Learning

In reinforcement learning, an agent learns to make decisions by taking actions in an environment to maximize cumulative rewards [SB18]. The learning process is inspired by behavioural psychology, where the agent learns from the consequences of its actions rather than from a fixed dataset. Applications include the following.

Q-Learning. A model-free reinforcement learning algorithm that seeks to find the best action to take given the current state, by learning a value function that predicts the expected utility of actions [WD92].

Policy Gradient Methods. These methods directly learn a policy that maps states to actions, optimizing the policy through gradient ascent on expected rewards. They are widely used in continuous action spaces [SB18].

Machine learning's versatility and effectiveness have made it a cornerstone of modern AI, enabling systems to tackle a wide array of tasks from classification and prediction to decision-making and pattern recognition. As ML continues to evolve, it will likely drive further advancements in AI across multiple domains.

2.3.4 Deep Learning

Deep Learning, a specialized subset of machine learning, has garnered considerable attention due to its remarkable success in a variety of domains, particularly those involving large-scale data and intricate pattern recognition [LBH15]. Unlike traditional machine learning methods, which often rely on manual feature extraction, deep learning models automatically learn hierarchical representations of data through multiple layers of abstraction. These models are built on artificial neural networks, inspired by the human brain's architecture, where interconnected neurons process information through weighted connections. The capacity of deep learning models to learn complex and non-linear patterns has made them exceptionally effective for tasks such as image recognition, natural language processing, and speech recognition [GBC16].

2.3.4.1 Deep Neural Networks

A Deep Neural Network (DNN) is an extension of a basic artificial neural network, characterized by the presence of multiple hidden layers between the in-

put and output layers [LBH15]. Each hidden layer in a DNN is composed of numerous neurons, each performing a weighted sum of its inputs followed by a non-linear activation function. These hidden layers enable the network to capture intricate patterns and representations within the data, which are often difficult to detect using shallow models [LBH15]. The depth of the network allows it to learn from the data in a hierarchical fashion, where each successive layer builds upon the features learned by the previous one. This layered architecture makes DNNs particularly powerful for a wide range of complex tasks, including but not limited to image classification, natural language understanding, and autonomous driving [LBH15].

2.3.4.2 The Backpropagation Algorithm

The effectiveness of deep learning models can be largely attributed to the backpropagation algorithm [RHW86], a critical method for training neural networks. Backpropagation, short for "backward propagation of errors," is an iterative optimization algorithm used to minimize the loss function by updating the model's weights. The process involves two main phases: forward propagation and backward propagation.

During forward propagation, the input data is passed through the network to generate an output, which is then compared against the true target values to calculate the loss. In backward propagation, this loss is propagated back through the network, and the gradient of the loss with respect to each weight is computed. These gradients are then used to update the weights in a direction that minimizes the loss, typically by using a gradient descent optimization algorithm. The repeated application of these steps across multiple iterations (epochs) allows the network to iteratively refine its weights, thereby improving its performance on the task [RHW86].

The backpropagation algorithm has been pivotal in the development of deep learning, as it enables the efficient training of deep networks by systematically reducing the prediction error [LBBH98]. The use of gradient descent in conjunction with backpropagation ensures that the network converges to a set of weights that optimally fit the training data, making it possible to generalize well to new, unseen data. This algorithm remains a fundamental technique in the training of neural networks, despite the emergence of more sophisticated optimization methods in recent years [GBC16].

Algorithm 4 Backpropagation Algorithm

Require: Training data $(x^{(i)}, y^{(i)})$, learning rate η
Ensure: Trained weights $\mathbf{W}$
Initialize weights $\mathbf{W}$ randomly
for each epoch **do**
  for each training example $(x^{(i)}, y^{(i)})$ **do**
    Forward propagation: Compute the predicted output $\hat{y}^{(i)} = f(x^{(i)}; \mathbf{W})$
    Compute the loss: $\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$
    Backward propagation: Compute gradients $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$
    Update weights: $\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}}$
  end for
end for
return $\mathbf{W}$

2.3.4.3 Advanced Techniques in Deep Learning

Deep learning has grown far beyond its initial implementations of simple feed-forward neural networks, incorporating a variety of sophisticated techniques designed to enhance model performance, reduce training time, and improve generalization to unseen data. Below are some of the most influential techniques in modern deep learning.

Convolutional Neural Networks (CNNs). Convolutional Neural Networks (CNNs) are a class of deep neural networks that have proven highly effective in processing data with a grid-like topology, such as images. CNNs use a mathematical operation called convolution in place of general matrix multiplication in at least one of their layers [LBBH98]. This operation allows the network to recognize spatial hierarchies in data, making CNNs particularly adept at tasks like image recognition and classification. LeNet-5, developed by Yann LeCun and others in 1998, is one of the earliest and most well-known CNNs, designed for handwritten digit recognition [LBBH98].

Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) Networks. Recurrent Neural Networks (RNNs) are designed to handle sequential data by maintaining a memory of previous inputs through internal states. This architecture makes RNNs suitable for tasks such as language modeling, translation, and time-series prediction [HS97]. However, standard RNNs suffer from the problem of vanishing gradients, which hampers their ability to capture long-term dependencies in data. To address this, Long Short-Term

Memory (LSTM) networks were introduced, incorporating mechanisms called gates that regulate the flow of information, allowing the network to maintain and retrieve long-term information effectively [HS97].

Generative Adversarial Networks (GANs). Generative Adversarial Networks (GANs), introduced by Ian Goodfellow et al. in 2014, is a groundbreaking approach in generative modelling [GPAM+14]. GANs consist of two neural networks—a generator and a discriminator—that are trained simultaneously through adversarial processes. The generator attempts to produce data indistinguishable from real data, while the discriminator tries to differentiate between real and generated data. This competition drives the generator to create increasingly realistic outputs, which has led to GANs becoming a powerful tool for applications such as image synthesis, video generation, and style transfer [GBC16].

Attention Mechanisms and Transformers. Attention mechanisms have become integral to many state-of-the-art deep learning models, particularly in the realm of natural language processing (NLP). Attention allows models to focus on specific parts of the input sequence when producing an output, which is crucial for tasks involving long sequences of data [VSP+17]. The Transformer model, introduced by Vaswani et al. in 2017, is built entirely on attention mechanisms, dispensing with recurrence entirely [VSP+17]. Transformers have since become the foundation for many cutting-edge models, including BERT (Bidirectional Encoder Representations from Transformers) [DCLT18] and GPT (Generative Pre-trained Transformer) [RNSS18].

Transfer Learning. Transfer learning is a technique where a model developed for a particular task is reused as the starting point for a model on a second task. This approach is especially useful when data for the second task is scarce [PY09]. Pre-trained models like BERT and GPT are often fine-tuned on specific tasks, enabling them to achieve high performance with relatively little data. Transfer learning has become a standard practice in NLP and computer vision, where models trained on large datasets (like ImageNet) are adapted for more specific tasks [YCBL14].

2.3.4.4 Challenges and Future Directions

Despite its successes, deep learning faces several challenges. One of the most significant is the need for large amounts of labelled data, which is often costly and time-consuming to obtain. Techniques like semi-supervised learning and unsupervised learning are being developed to mitigate this dependency by leveraging unlabeled data [KW14]. Another challenge is the interpretability of deep learning models; as models become more complex, understanding the rationale behind their predictions becomes increasingly difficult. Research into explainable AI (XAI) aims to make models more transparent and trustworthy [Gun17].

Furthermore, as models grow in size, the computational resources required to train them also increase dramatically. This trend raises concerns about the environmental impact of deep learning, prompting the development of more efficient models and training algorithms [SGM19].

In the future, advancements in areas such as quantum computing, neuromorphic computing, and advanced optimization techniques may further revolutionize deep learning, making it more powerful, efficient, and versatile across an even broader range of applications [Pre18, Fur16].

2.4 Machine Learning-Based SAT Solving

The integration of traditional SAT-solving techniques with modern machine-learning approaches has demonstrated significant potential in recent years. Learning-based SAT solvers, such as SATformer [SLK+22], leverage neural networks and reinforcement learning to predict variable assignments and uncover complex patterns within problem instances. These solvers dynamically refine their strategies based on the problem's characteristics, leading to more efficient handling of diverse SAT instances [YP19]. The incorporation of machine learning within CDCL solvers represents a frontier in SAT solving [Lia18]. Techniques like neural branching strategies and reinforcement learning-based adaptations have shown promise in reducing solving times and enhancing solver robustness across diverse problem instances [WHT+21, KGWC20]. These hybrid approaches allow solvers to dynamically tailor their behaviour and optimize search strategies in real time, which significantly improves performance by reducing the number of backtracks and speeding up the solver.

2.4.1 SAT Solver Augmentation

Augmenting traditional SAT solvers with machine learning techniques has emerged as a crucial development in computational logic. This approach focuses on enhancing various components of SAT solvers, including variable selection, heuristic learning, clause management, and configuration tuning.

2.4.1.1 Variable Selection

Variable selection plays a critical role in determining the efficiency of SAT solvers. Traditional heuristics, such as VSIDS[MM+01], often lack the adaptability needed for diverse problem instances. Machine learning models developed by Liang et al. [LOG+18] predict the most promising variables for branching based on features extracted from problem instances, leveraging also on the historical performance of variables in prior SAT instances. This approach demonstrated notable improvements in solver performance by reducing unnecessary computations and focusing the solver's efforts on more promising search paths. These models have been further refined by integrating adaptive heuristics that dynamically adjust based on the specific characteristics of the SAT instance, leading to substantial improvements in solving efficiency [BS22, UONW23].

2.4.1.2 Heuristic Learning

Beyond variable selection, machine learning is increasingly used to enhance heuristic learning, allowing solvers to dynamically adapt their strategies. Kurin et al. [KGWC20] proposed a hybrid approach combining reinforcement learning with traditional SAT-solving techniques, achieving a balance between computation time and solution quality. Yolcu and Poole [YP19] demonstrated that deep reinforcement learning can significantly improve solver performance by reducing conflicts and accelerating convergence. Additionally, Han [Han20] contributed to the field by applying machine learning to predict glue variables, optimizing heuristic selection dynamically. Their work focused on training models that could predict which heuristics would be most effective for different phases of the SAT solving process. By integrating these predictive models, SAT solvers could switch between heuristics dynamically, depending on the current problem characteristics, leading to a more adaptable and efficient solving process.

2.4.1.3 Clause Management

Machine learning has also been applied to optimize clause management in SAT solvers. Devriendt et al. [DBB17] introduced symmetric explanation learning, which predicts the usefulness of clauses based on their participation in past conflict resolutions. This method leverages symmetries in SAT problems to identify and retain clauses likely to be useful in future conflict resolutions. Similarly, Driel et al. [vDDYS21] applied machine learning to manage clauses in hybrid CP-SAT solvers, focusing on retaining only the most critical clauses, thereby reducing memory overhead and improving solving efficiency. Liang et al. [LOM+18] extended their machine learning applications to the management of clauses in SAT solvers. They proposed a model that dynamically evaluates clause usefulness based on their historical contribution to solving past instances. This allows the solver to retain only those clauses that are likely to contribute to resolving the current instance, thus optimizing memory usage and improving overall solver efficiency. Recent advancements have integrated machine learning models to predict clause usefulness dynamically, further enhancing the effectiveness of activity-based deletion [LOG+18].

2.4.1.4 Configuration Tuning.

The configuration of SAT solvers, including variable ordering heuristics and restart strategies, plays a crucial role in their overall performance. Traditionally, these configurations have been manually tuned; however, machine learning offers a more systematic approach. The Machine Learning for Combinatorial Optimization (ML4CO) competition demonstrated that data-driven methods could surpass traditional heuristics in solver configuration [GBC+22]. Additionally, Biedenkapp et al. introduced dynamic algorithm configuration using reinforcement learning, enabling solvers to adjust configurations online based on problem-specific features [BDKH20]. These advancements underscore the potential of machine learning to enhance SAT solver efficiency by optimizing configurations dynamically during the solving process.

2.4.1.5 Impact of Machine Learning on SAT Solvers

The integration of machine learning into SAT solving has had a profound impact across various solver components:

- **Efficiency:** Machine learning models have streamlined variable selection,

heuristic learning, and clause management, leading to faster solving times and reduced memory overhead [WHT+24].

- **Adaptability:** By dynamically adjusting strategies based on problem characteristics, machine learning-enhanced solvers can handle a broader range of SAT instances with greater robustness [SYZ+24].
- **Scalability:** The ability to automatically tune configurations allows solvers to scale more effectively to larger and more complex problems, as evidenced by recent competitions and benchmark studies [SLB+21].

Overall, the use of machine learning in SAT solvers has not only enhanced their performance but has also expanded their applicability in real-world scenarios, including hardware verification, AI planning, and cryptographic analysis.

2.4.2 Deep Learning and SAT

The intersection of deep learning and SAT solving has become a promising and rapidly advancing field in computational logic [BL17]. Deep learning, particularly through neural networks, offers powerful tools for capturing complex patterns and relationships within SAT instances, significantly enhancing the efficiency and effectiveness of SAT solvers.

2.4.2.1 End-to-End Learning for SAT Solving

End-to-end learning approaches aim to solve SAT problems by training models to predict satisfiability directly from raw data without relying on handcrafted features [SLB+18]. This methodology enables the models to learn useful representations autonomously, making them adaptable to various SAT problems.

Bünz and Lamm [BL17] pioneered the use of Graph Neural Networks (GNNs) for SAT solving by representing Boolean formulas as graphs, allowing the network to learn the underlying relational structures. Guo et al.[GZL+23] reviewed the evolution of machine learning-enhanced SAT solvers, highlighting models like NeuroSAT, which employs GNNs for SAT solving in an end-to-end manner [SB19]. SATformer, introduced by Shi et al. [SLK+22], utilizes a Transformer-based architecture to better understand clause relationships, significantly improving solver performance on complex SAT instances.

2.4.2.2 Graph Neural Networks in SAT Solving

GNNs have demonstrated significant potential in addressing the relational complexities of SAT problems. By representing variables and clauses as nodes in a graph, GNNs effectively capture dependencies critical to solving SAT problems.

Yan et al. [YLS+23] extended the capabilities of GNNs by integrating Recurrent Neural Networks (RNNs) to improve predictions for variable assignments, creating a hybrid model known as AsymSAT. This integration helps solve more complex SAT instances by leveraging both the spatial relationships captured by GNNs and the temporal dependencies handled by RNNs. Yu et al. [YLZ19] introduced G2SAT, a deep generative framework that learns to generate SAT formulas similar to real-world instances, demonstrating the versatility of GNNs beyond satisfiability checking.

2.4.2.3 Neural Networks as Standalone SAT Solvers

Neural networks have also been explored as standalone solvers for SAT problems, particularly in specialized or highly complex cases where traditional methods may not be as effective. Marino [Mar21] developed a deep learning algorithm for solving the Maximum Exact 3-Satisfiability problem, showcasing how neural networks can be tailored to specific SAT variants. Yang et al. [YWC+19] extended this by exploring GNN-based solvers for Quantified Boolean Formulas (QBF), demonstrating that neural networks can also handle more complex logical structures. Fournier et al. [FLC+22] utilized deep reinforcement learning with adversarial GNNs to create a heuristic that outperforms traditional solvers, showing the potential of neural networks in enhancing SAT solving efficiency.

2.4.2.4 Innovative Neural Network Architectures for SAT

Various neural network architectures have been proposed to address the unique challenges of SAT problems. These architectures leverage deep learning to model the intricate structures and relationships inherent in SAT instances. Narodytska et al. [NKR+18] introduced an exact Boolean representation of binarized deep neural networks, integrating SAT solvers for verification tasks. This approach highlights the synergy between deep learning and SAT solving in ensuring model correctness. Pinkas and Cohen [PC17] developed neural architectures that encode structured relational knowledge, enhancing the ability of neural networks to reason about SAT problems. Azizan et al. [ASA+23] pro-

posed a hybrid network combining 3-Satisfiability structures with fuzzy logic and metaheuristic algorithms, demonstrating the potential for hybrid models to solve SAT problems more efficiently.

2.4.3 Challenges and Future Directions

While integrating machine and deep learning with SAT solving shows significant promise, challenges remain, particularly regarding scalability and generalization across different SAT problem types. The continued development of hybrid models that combine the strengths of traditional SAT solvers with learning approaches is likely to be a key area of future research, aiming to create more robust and efficient solvers capable of tackling increasingly complex SAT problems.

Chapter 3

Machine Learning Predictions based on SAT Features

Abstract. *This chapter explores the integration of machine learning techniques with SAT problems. By leveraging features extracted from SAT instances, including structural and statistical properties, we train machine learning algorithms to predict satisfiability, problem category, and model count. A key contribution is the integration of multiple feature sets and their application across diverse tasks. The findings highlight the potential of ML to improve solver performance through effective feature utilisation and predictive modelling. This chapter is based on our previous publication in* ***SOCS*** *and* ***ICTAI****: **SAT Feature Analysis for Machine Learning Classification Tasks** [DPVO23] and **A Machine Learning Approach to Model Counting** [DVO24], as well as the* ***arXiv*** *paper: **SATfeatPy–A Python-based Feature Extraction System for Satisfiability** [PDVO22] and the code is available.* [1]

3.1 Introduction

The integration of machine learning (ML) techniques with SAT solving represents a promising avenue for addressing some of the longstanding challenges in the field. This chapter focuses on the application of ML models to predict the satisfiability, problem category, and model count of SAT instances using

[1] https://github.com/mardalla/SATfeatPy

features extracted from these instances. The problem of predicting satisfiability (SAT/UNSAT) involves determining whether a given Boolean formula has a satisfying assignment. This task is not only computationally intensive but also highly dependent on the instance's specific structure and statistical properties. Similarly, categorising SAT instances into predefined problem classes can provide valuable insights into the nature of the instances, enabling more targeted and efficient solving strategies. Specifically, in this research, the multiclass classification problem involves categorising SAT instances into distinct problem families: k-Clique, k-Coloring, Clique Coloring, Dominating Set, Matching Principle, Ordering Principle, Subset Cardinality, Tiling, Tseitin, and Pigeonhole Principle formulas. Each of these categories represents unique structural and logical constraints common in industrial and combinatorial applications. Finally, estimating the model count, or the number of satisfying assignments, is crucial for probabilistic reasoning and quantitative analysis applications. Each of these tasks benefits from the extraction and analysis of meaningful features from SAT instances. However, solving these tasks is computationally intensive due to their inherent complexity—predicting satisfiability is NP-complete and model counting is a #P-complete problem. The substantial time and resources required to solve them highlight the need for alternative approaches, such as using ML models to predict these properties based on instance features. Feature extraction is a critical step in applying ML to SAT solving, which is needed due to the variability in instance length, symmetries, and other structural properties. The features used in this research are derived from established methodologies, including those implemented in SATzilla [XHHLB08], the work of Ansótegui et al. [ABGCL17], and Alfonso et al. [AM14].

The SATzilla features, utilised in the well-known portfolio-based SAT solver SATzilla, include a comprehensive set of metrics to describe SAT instances, such as problem size, variable-clause graph statistics, balance features, proximity to Horn formulas, and results from DPLL and local search probing. These features capture various aspects of the SAT instances, such as their structural complexity and empirical hardness. The Ansótegui features introduce additional structural metrics based on graph representations, such as the power-law exponent, modularity, and fractal dimension of the variable incidence graph (VIG) and clause-variable incidence graph (CVIG). These metrics provide insights into the inherent structural properties of the instances, which can significantly impact their solvability. Finally, the Alfonso features expand this approach by introducing several weighted graphs and associated statistical measures, including

clause-variable graphs, resolution graphs, and binary implication graphs. These graphs capture intricate relationships between variables and clauses, offering a more nuanced understanding of the formula's structure.

This research differs from the state of the art by integrating multiple feature sets and applying them across various ML tasks within the SAT domain. By treating SAT as an ML task, we aim to develop predictive models that can effectively use features extracted from SAT instances to make accurate predictions for satisfiability, problem categorisation, and model counting. These tasks cover different aspects of supervised ML: binary classification (satisfiability prediction), multi-class classification (problem categorisation), and regression (model counting). The goal is to create features that preserve relevant information for decision-making, ultimately enhancing solver performance and efficiency.

This chapter contributes to the ongoing efforts to integrate ML with SAT solving by comprehensively analysing feature sets and their impact on prediction tasks [DO08]. In our research, we combined multiple established feature extraction methodologies and applied them to various ML tasks within the SAT domain. Additionally, we introduce the SATfeatPy library [PDVO22], which offers a unified implementation of these feature extraction techniques in Python, facilitating further research and prototyping in this area.

3.2 Dataset Description

The use of meaningful datasets is critical for evaluating the machine learning models developed for predicting the satisfiability, category, and model count of SAT instances. The datasets include various problem types that reflect the challenges encountered in industrial SAT applications. The primary motivation for generating our datasets was the need for a large number of instances with controlled variables and clauses. Machine learning algorithms heavily rely on the quality and representativeness of the training datasets. These datasets will be described in depth in the following subsections and will be featured in the further chapter of this thesis.

3.2.1 Dataset for Satisfiability and Category Prediction

The dataset comprises 3000 CNF instances, evenly split between SAT and UNSAT problems. These instances are derived from various families of problems

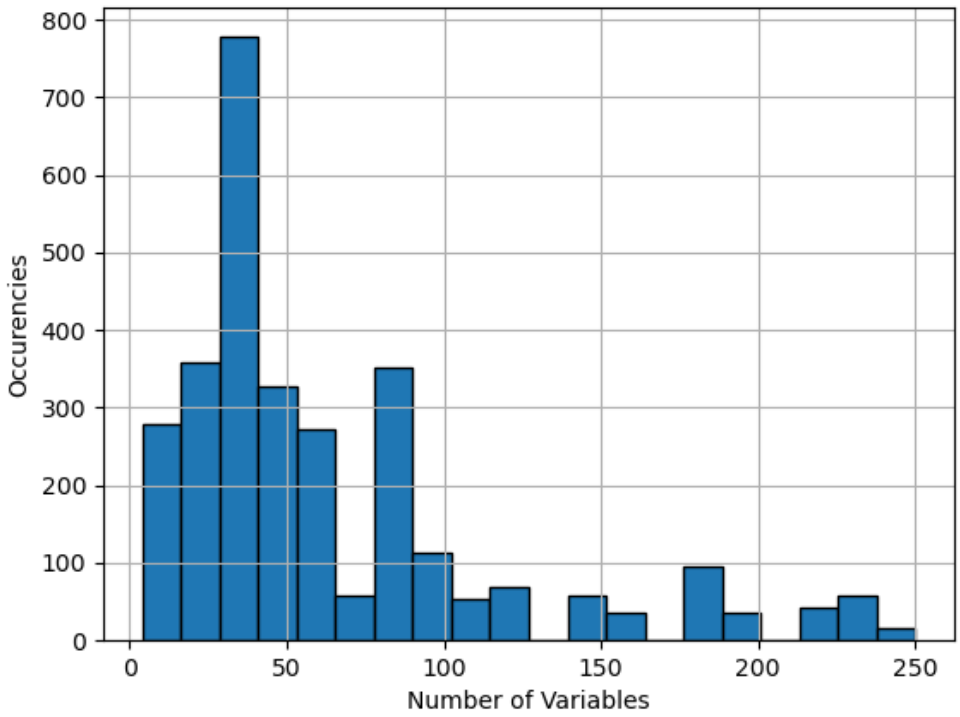


Figure 3.1: Histogram of number of variables for the dataset

that are commonly found in industrial contexts. Each problem type presents unique structural and statistical properties, offering a comprehensive testbed for the machine learning models. In Fig. 3.1, 3.2 the histograms of the number of variables and clauses for the dataset are shown.

3.2.1.1 Problem Families

The dataset includes instances from the following families:

1. **k-Clique Problems ($k = 3, 4, 5$):** These problems involve finding a complete subgraph (clique) of size k within a given graph. For a given graph $G = (V, E)$, a k-clique is a subset of k vertices such that every pair of vertices in the subset is connected by an edge. The k-clique problem is NP-complete and serves as a benchmark for testing the combinatorial complexity that SAT solvers can handle[Bol76, DF95, CKJ08].

2. **k-Coloring Problems ($k = 3, 4, 5$):** Graph coloring problems involve assigning colors to the vertices of a graph such that no two adjacent vertices share the same color. The 3, 4, and 5 coloring problems specifically refer to instances where three, four, or five colors are used, respectively.

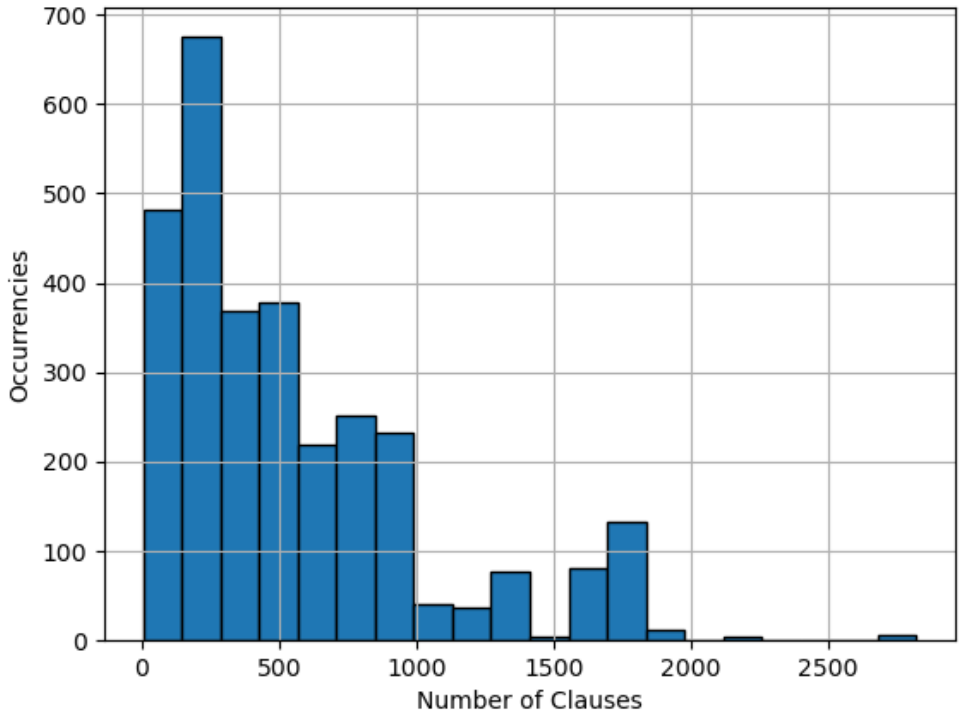


Figure 3.2: Histograms of number of clauses for the dataset

These problems are crucial for testing the ability of SAT solvers to handle constraints related to adjacency and graph structure [Tut59, Wes00].

3. **Clique Coloring Problems:** These problems combine elements of cliques and graph coloring. The goal is to color a graph such that every clique of a certain size is monochromatic (all vertices in the clique share the same color). This type of problem is particularly challenging because it merges the difficulty of finding cliques with the constraints of coloring [GJ79].

4. **Dominating Set Formulas:** A dominating set in a graph is a subset of vertices such that every vertex in the graph is either in the dominating set or adjacent to a vertex in the dominating set. Finding a minimum dominating set is a classical NP-complete problem, often encoded as a SAT problem to leverage the efficiency of SAT solvers [CKJ08].

5. **Matching Principle Formulas:** These problems are based on the matching principle in bipartite graphs, where the goal is to find a matching that covers as many vertices as possible. This type of problem is widely applicable in network design and resource allocation, making it a valuable benchmark for SAT solvers [Tar72].

6. **Ordering Principle Formulas:** Ordering principle problems involve arranging a set of items in a specific order under given constraints. These problems can be encoded as SAT problems where the constraints are represented as clauses that enforce the ordering conditions [GJ79].
7. **Subset Cardinality Formulas:** These problems involve constraints on the cardinality (size) of subsets within a larger set. They are often used to model problems in areas such as combinatorial optimization and resource allocation [GJ79].
8. **Tiling Formulas:** Tiling problems involve covering a plane or space with tiles according to specific rules without gaps or overlaps. These problems are complex and are used to test the spatial reasoning capabilities of SAT solvers [GJ79].
9. **Tseitin Formulas:** Tseitin formulas are based on Tseitin's transformation, which is used in the context of propositional logic and circuit design. These formulas are challenging for SAT solvers due to the large search space required to find a solution [Tse83].
10. **Pigeonhole Principle Formulas:** The pigeonhole principle states that if more items than containers are placed in the containers, at least one container must contain more than one item. Encoding this principle as a SAT problem involves creating clauses that enforce this condition, making it a useful benchmark for SAT solvers [Sto76].

3.2.1.2 Instance Distribution and Characteristics

The dataset is carefully balanced, with half of the problems being satisfiable and the other half unsatisfiable as well as equally distributed between 10 categories. This balance is crucial for training machine learning models that need to differentiate between SAT and UNSAT instances and categories effectively. The number of variables and clauses in the instances varies significantly, ranging from 4 to 250 variables and 4 to 2820 clauses. This variety ensures that the models are exposed to a wide range of problem sizes and complexities, which is essential for robust evaluation.

3.2.2 Dataset for Model Counting

We focused on smaller instances in terms of variables and clauses because the approximate model counter requires significant computation time, and the

graphs needed for computing some statistical features have large memory footprints. Additionally, we required a smooth distribution for the natural logarithm of the number of solutions for the instances, as this is the target metric for different model counters in Model Counting competitions. We decided to generate two datasets, and assembled two further ones to test the algorithms generalisation properties.

3.2.2.1 Dataset 1

The first dataset was generated using the tool described in [EO22]. This tool produces propositional formulas in the CNF DIMACS format, which were used as benchmarks during the International Model Counting Competition [FHH21]. The generator is highly parameterisable, allowing control over the number of variables, clauses, and literals in each clause. The generated instances have been tested against other challenging model counting instances on state-of-the-art model counters, showing that the smallest unsolved instances of the competition were produced using this method. The tool generates three types of instances: completely random k-CNF instances, random CNF instances with exactly one solution, and random instances with an added cluster of solutions. Detailed algorithm descriptions can be found in [EO22]. The distribution of the solution is shown in 3.3.

3.2.2.2 Dataset 2

The second dataset was generated using the procedure explained in [SB22]. This dataset was employed to compare our machine learning algorithms with the deep learning methods used by the authors. The dataset consists of instances drawn from a random distribution, with clauses and variables sampled within a fixed interval. Each clause has an average of five variables, negated with a 50% probability. The number of solutions was provided as a target for the learning algorithm and was computed using Cachet [SBB$^+$04] and Sharpsat [Thu06] model counters. The distribution of the solution is shown in 3.4.

3.2.2.3 Dataset 3

The third dataset consists of 2031 publicly available instances from diverse applications of model counting, including probabilistic reasoning, plan recognition, DQMR networks, ISCAS89 combinatorial circuits, quantified information flow, logistics, and more [BBK89, SBK05, CMV16, LLM16]. Out of the 2031 we

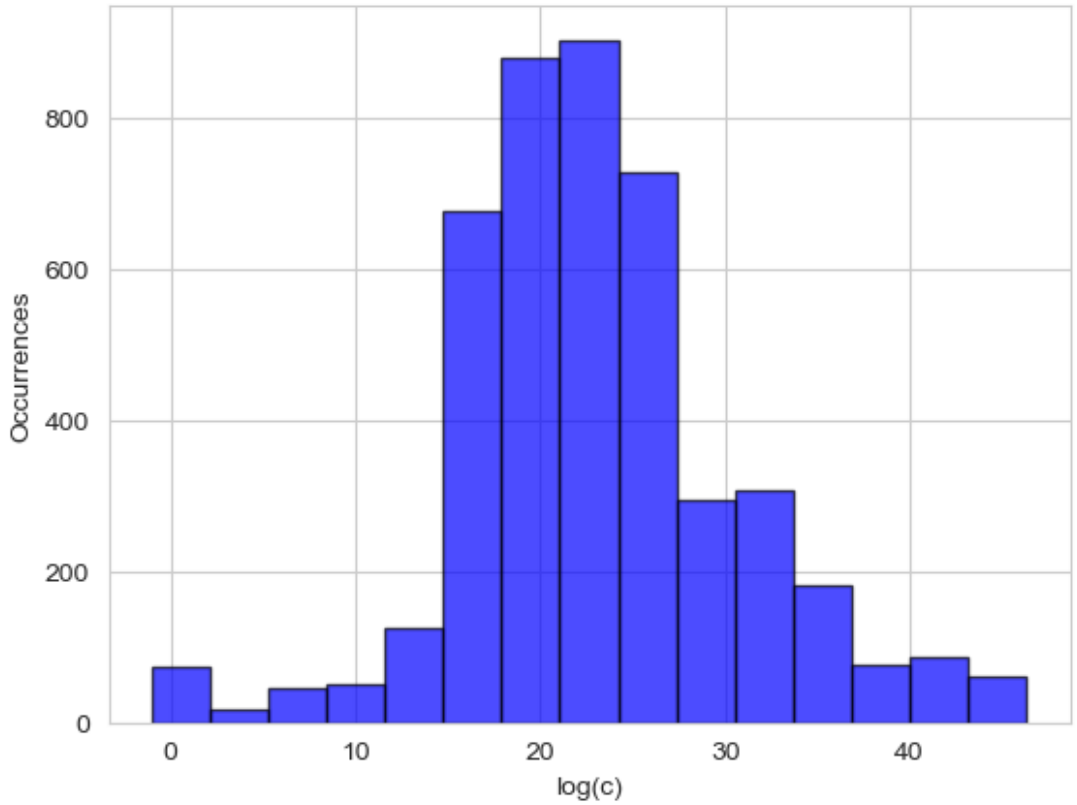


Figure 3.3: Natural logarithm distribution of the number of solutions for the instances of Dataset 1

have been able to provide an exact model count only for 1597, using *d4*, *gpmc* and *Cachet* [FHH21]. This was due to our limited computing resources that consisted in a *Intel(R) Xeon(R) Gold 6240 CPU 2.60GHz* and *256 GB* of RAM memory. These benchmarks have previously been utilized in model counting studies, ensuring a comprehensive and challenging testbed for assessing our learning algorithm scalability and efficiency performances. The distribution of the solution is shown in 3.5.

3.2.2.4 Dataset 4

The fourth dataset comprises instances used in the Model Counting Competition. These problems were provided by the organizers along with the true value of the model count in the common logarithm. While we do not have detailed information on how these instances were produced, we included this dataset to test our algorithms' generalization ability [FHH21]. The distribution of the solution is shown in 3.6.

The main statistical properties of the three datasets are summarized in Table

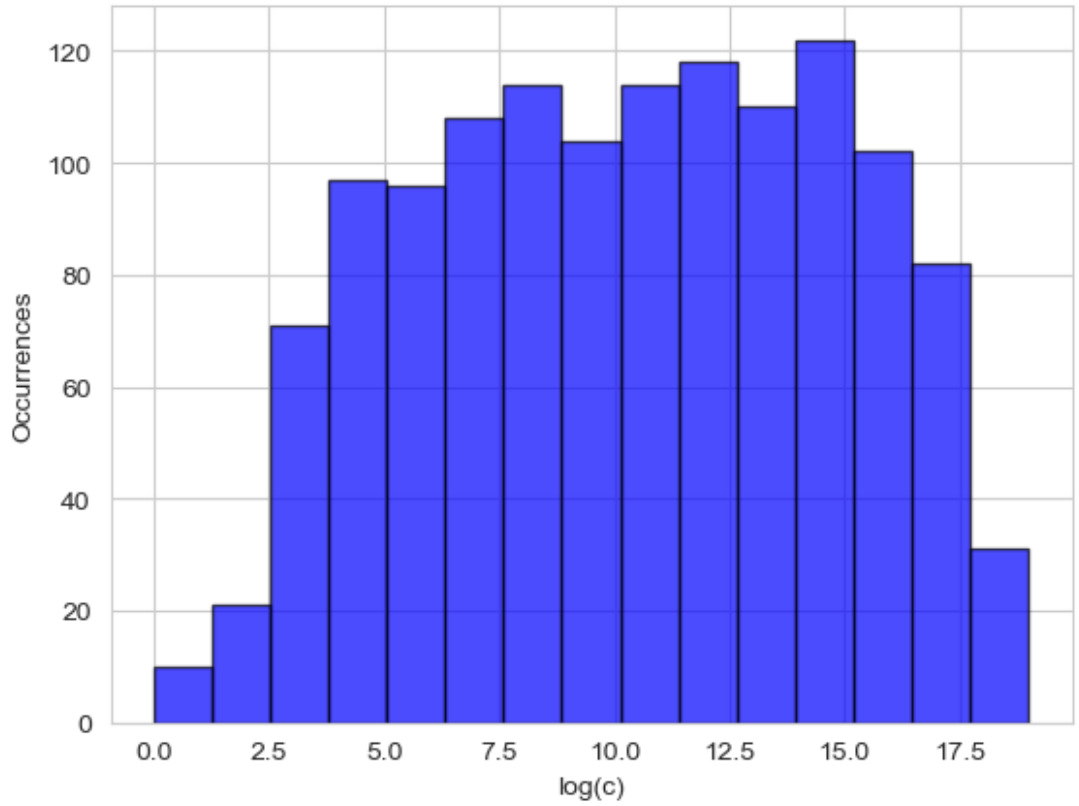


Figure 3.4: Natural logarithm distribution of the number of solutions for the instances of Dataset 2

Table 3.1: Statistics for Dataset 1 and Dataset 2. The value -1 indicates a #SAT problem without solutions, i.e., an unsatisfiable instance

	Dataset 1		Dataset 2	
	Train	Test	Train	Test
# of instances	3000	450	1300	300
# of clauses	[40−500]	[40−500]	[20−50]	[20−50]
# of variables	[40−100]	[40−100]	[10−30]	[10−30]
Mean(log(c))	23.04	23.16	10.30	9.86
Range(log(c))	[-1−47.8]	[-1−46.6]	[0−18.1]	[0−17.5]

3.1 and 3.2.

3.3 Feature Sets

In this section, we provide a comprehensive dissertation on the feature sets introduced by Xu et al. [XHHLB08], Ansotegui et al. [ABGCL17], and Alfonso et al. [AM14], which are extracted to obtain information on the structural,

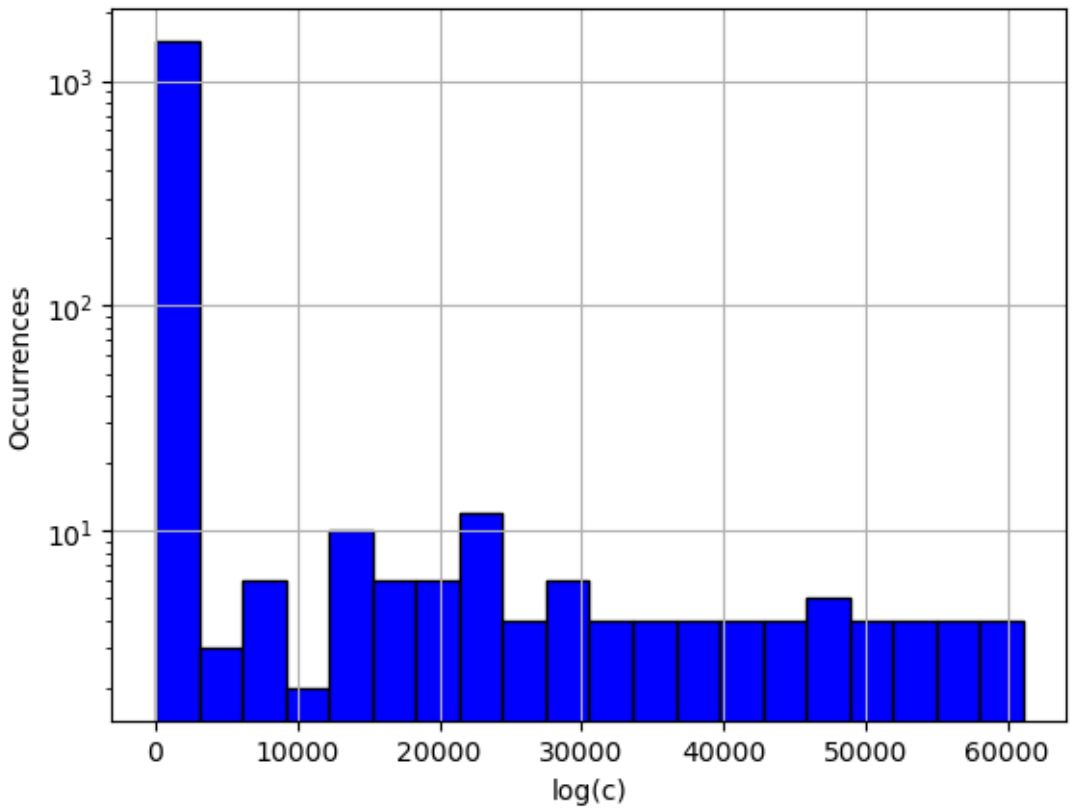


Figure 3.5: Natural logarithm distribution of the number of solutions for the instances of Dataset 3

Table 3.2: Statistics for Dataset 3 and Dataset 4. The value -1 indicates a #SAT problem without solutions, i.e., an unsatisfiable instance

	Dataset 3		Dataset 4
	Train	Test	
# of instances	300	30	1597
# of clauses	[49−274818]	[110−72344]	[10−$2.06{\cdot}10^6$]
# of variables	[18−2815]	[68−2958]	[15−$3.31{\cdot}10^5$]
Mean(log(c))	59.11	55.43	1971
Range(log(c))	[-1−649.2]	[-1−276.3]	[0.30−66136]

statistical, and empirical properties of SAT instances. These feature sets enable the application of machine learning models to predict the satisfiability, category, and model count of SAT instances effectively.

3.3.1 SATzilla Features

SATzilla is a well-known portfolio-based SAT solver that employs a wide range of features to characterize SAT instances [XHHLB08]. These features include

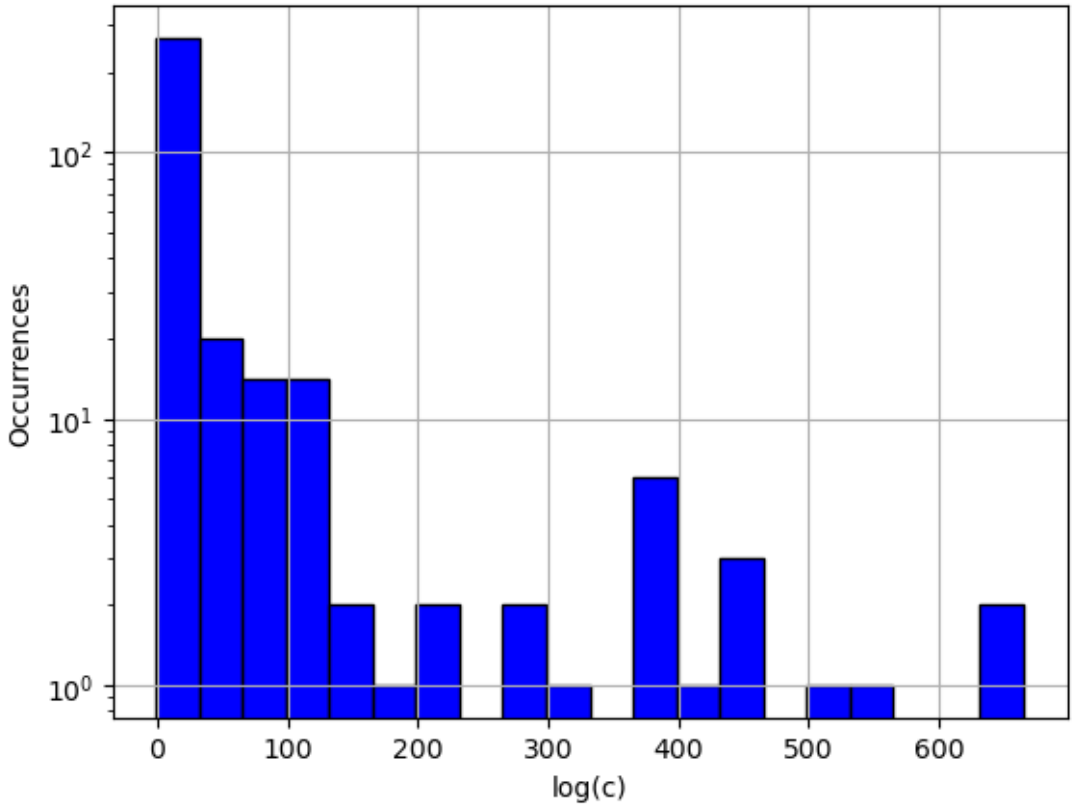


Figure 3.6: Natural logarithm distribution of the number of solutions for the instances of Dataset 4

structural, statistical, and probing characteristics that provide a detailed representation of the instances. The feature set used by SATzilla can be broadly categorized into the following groups:

3.3.1.1 Problem Size

Features related to the size of the SAT problem, including the number of variables and clauses. These basic metrics provide an initial understanding of the problem's complexity.

- **Number of Clauses:** Represents the total count of clauses in the SAT instance, providing a direct measure of the instance's size.
- **Number of Variables:** Denotes the total number of unique variables present in the SAT instance, indicative of the problem space's breadth.
- **Ratio of Clauses to Variables:** This ratio, along with its transformations (square, cube, reciprocal, etc.), provides nuanced insights into the problem's complexity. For instance, the absolute value of the ratio minus a

specific constant (e.g., 4.26) and its higher-order transformations are also considered.

3.3.1.2 Variable-Clause Graph (VCG)

This graph represents the bipartite relationship between variables and clauses. Key features derived from the VCG include the degree statistics of variables and clauses, such as mean, standard deviation, and entropy. These statistics help in understanding the connectivity and structure of the formula.

- Degree statistics of variable nodes, capturing metrics like mean, median, maximum, minimum, and standard deviation.
- Degree statistics of clause nodes, reflecting the distribution of clause sizes.
- Graph density, indicating the interconnectedness of variables and clauses.
- Connected components, highlighting isolated subproblems within the SAT instance.

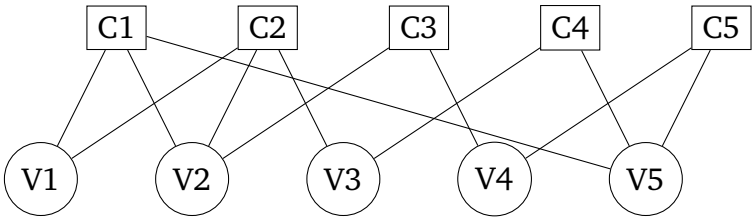


Figure 3.7: Example of a more complex Variable-Clause Graph (VCG). Circles represent variables and rectangles represent clauses.

3.3.1.3 Variable Graph (VG)

In the variable graph, nodes represent variables, and edges represent occurrences of these variables in the same clause. Features extracted from the VG include node degree statistics, which provide insights into how variables interact within the problem.

- Degree statistics, including mean, median, maximum, minimum, and standard deviation.
- Graph diameter, reflecting the reachability and spread of interactions.
- Clustering coefficient, measuring the tendency of variable interactions to cluster.

- Centrality measures, such as eigenvector and betweenness centrality, indicating the influence of variables within the VG.

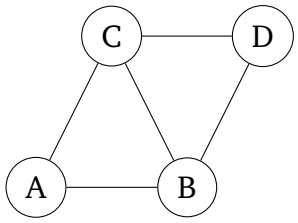


Figure 3.8: Example of a Variable Graph (VG) representation. Nodes represent variables and edges represent co-occurrences in clauses.

3.3.1.4 Balance Features

These features measure the balance between positive and negative literals in the formula. Metrics such as the mean, variation coefficient, and entropy of the ratio of positive to negative literals are included. A high imbalance can indicate a skewed problem structure, which may affect the performance of SAT solvers.

- **Literal Balance:** The ratio of positive literals to negative literals in the SAT instance.
- **Clause Balance:** The average balance of literals within each clause.
- **Variable Balance:** The distribution of positive and negative occurrences for each variable.

3.3.1.5 Bias Measurement

Evaluates the imbalance between positive and negative literals. It's calculated as:

$$\text{Bias} = 2 \times |0.5 - \frac{pos + neg}{total}| \tag{3.1}$$

where pos and neg denote the count of positive and negative literals, respectively.

3.3.1.6 Proximity to Horn Formula

These features assess how close the formula is to being a Horn formula [CN10], which is a specific type of formula with at most one positive literal per clause. Horn formulas are generally easier to solve, and this proximity measure can help predict problem difficulty.

- **Fraction of Horn Clauses:** The ratio of Horn clauses to the total number of clauses in the SAT instance.
- **Variable Occurrence in Horn Clauses:** Statistics such as mean, median, maximum, minimum, and standard deviation of the number of times each variable appears in Horn clauses.

3.3.1.7 DPLL Probing

DPLL (Davis-Putnam-Logemann-Loveland) probing features involve running the DPLL algorithm on the instance for a short period and recording various statistics from the search tree, such as the number of decisions, conflicts, and propagations encountered.

- **Unit Propagations:** The number of unit propagations at various depths, such as 1, 4, 16, 64, and 256, can provide insights into the complexity and structure of the SAT instance.
- **Backtracking Depths:** Features such as the average, maximum, and minimum backtrack depths can be extracted.
- **Search Tree Size Estimate:** Derived from the mean depth to contradiction and an estimate of the logarithm of the number of nodes in the search tree.

3.3.1.8 Local Search Probing

Local search probing features involve applying local search algorithms like GSAT and SAPS to the instance and recording statistics such as the number of flips and the best score achieved. These features provide empirical data on the instance's landscape and search difficulty.

- **Objective Function Value at Deepest Plateau (BestSolution):** Captures the value of the local search objective function at the deepest plateau reached during the search trajectory.
- **Steps to Deepest Plateau (BestStep):** Measures the number of steps taken by the local search algorithm to reach the deepest plateau.
- **Search Space Topology:** Features derived from GSAT and SAPS behaviors provide insights into the large-scale topological features of the local search space.

SATzilla's extensive feature set, which includes a total of 69 features, allows it to effectively classify and solve a wide range of SAT instances by leveraging the strengths of different solvers based on the instance characteristics [XHHLB08].

3.3.2 The Ansotegui et al. Features

This feature set, introduced by Ansótegui et al. [ABGCL17], focuses on structural features derived from the graph representations of SAT instances. These features aim to capture the underlying structure of the instances, which can be crucial for solving them efficiently. The key structural features included in the ANT set are:

3.3.2.1 Power Law Exponent

This feature measures the distribution of variable occurrences in the formula. Many real-world SAT problems exhibit a power-law distribution in their variable occurrences, where a few variables appear very frequently while most appear rarely. The power law exponent quantifies this distribution and helps in understanding the scale-free nature of the problem's structure.

3.3.2.2 Modularity

Modularity measures the extent to which a graph can be divided into modules or communities with dense connections internally and sparser connections between them. High modularity in the Variable Incidence Graph (VIG) indicates a strong community structure, which can affect the performance of SAT solvers by highlighting clusters of variables that interact more closely.

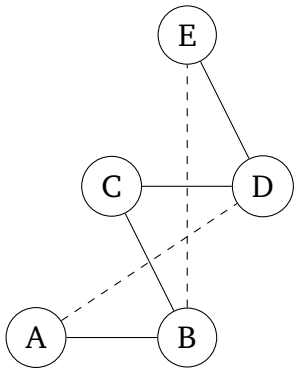


Figure 3.9: Example of a Variable Incidence Graph (VIG) representation with high modularity. Solid lines indicate strong internal connections within modules, while dashed lines indicate sparser connections between modules.

3.3.2.3 Fractal Dimension

The fractal dimension is a measure of the complexity and self-similar structure of the Variable Incidence Graph (VIG) and Clause Variable Incidence Graph (CVIG). It quantifies how the graph's detail changes with scale, providing insights into the hierarchical nature of the problem's structure. A high fractal dimension indicates a more complex and potentially harder problem.

These features are designed to enhance the ability of machine learning models to classify and solve SAT instances by capturing essential structural properties that influence solver performance [ABGCL17].

3.3.3 The Alfonso et al. Features

This feature set, proposed by Alfonso et al. [AM14], introduces a comprehensive set of features based on various graph representations and statistical measures. These features aim to provide a detailed and nuanced characterization of SAT instances. The main components of the feature set include:

3.3.3.1 Clause Variable Graphs (CV+ and CV-)

These graphs connect literals with the clauses in which they appear. The CV+ graph includes positive literals, while the CV- graph includes negative literals. Features extracted from these graphs include node degree statistics and connectivity measures, providing insights into the interaction patterns of literals.

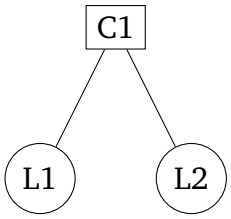


Figure 3.10: Example of a Clause Variable Graph (CVG) representation. Circles represent literals and rectangles represent clauses.

3.3.3.2 Variable Graph (VG)

Similar to the SATzilla VG, this graph represents variables and their co-occurrences in clauses, with additional weight considerations. Features from this graph include degree distributions and centrality measures, highlighting important variables within the problem.

3.3.3.3 Clause Graph (CG)

The clause graph connects clauses that share literals. This representation helps in understanding the interdependencies between clauses. Features include clustering coefficients and path lengths, which indicate the density and reachability within the graph.

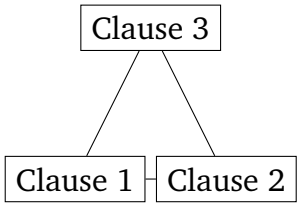


Figure 3.11: Example of a Clause Graph (CG) representation. Nodes represent clauses and edges represent shared literals between clauses.

3.3.3.4 Resolution Graph (RG)

The resolution graph connects clauses that can be resolved to produce a non-tautological clause. Features from this graph include node degrees and resolution path lengths, which provide insights into the potential simplifications within the formula.

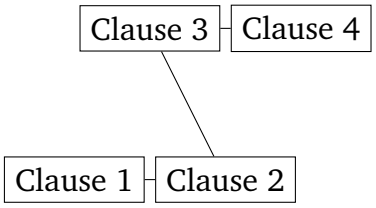


Figure 3.12: Example of a Resolution Graph (RG) representation. Nodes represent clauses and edges represent resolution paths.

3.3.3.5 Binary Implication Graph (BIG)

The BIG contains literals as vertices and binary clauses as edges, representing direct implications between literals. Features include strongly connected components and cycle detection, which help in understanding the propagation of implications within the problem.

3.3.3.6 AND/BAND/EXO Gate Graphs

These graphs represent relations between literals in specific logical gates (AND, BAND, EXO). Features include gate counts and connectivity measures, which

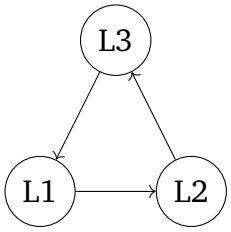


Figure 3.13: Example of a Binary Implication Graph (BIG) representation. Nodes represent literals and directed edges represent implications.

are crucial for understanding the logical structure and constraints within the formula.

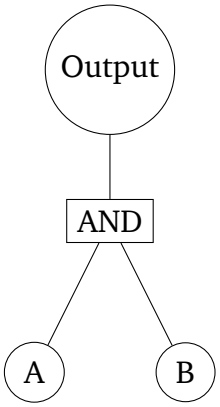


Figure 3.14: Example of an AND Gate Graph representation. Nodes represent literals and gates, and edges represent connections.

3.3.3.7 Recursive Weight Heuristic

The Recursive Weight Heuristic assigns a score to each literal, indicating its likelihood of being present in a model of the formula. The heuristic is refined over multiple iterations to improve the accuracy of the scores. The score $S(l)$ for a literal l is computed iteratively until convergence.

These features set provides a detailed characterization of SAT instances, capturing both local and global structural properties that are crucial for predicting solver performance and instance difficulty [AM14].

3.3.4 SATfeatPy: A Unified Feature Extraction Framework

A key contribution of this thesis is the development of SATfeatPy, a Python-based library that integrates and extends the feature extraction methodologies introduced by SATzilla [XHHLB08], Ansótegui et al. [ABGCL17], and Alfonso et al. [AM14]. This library consolidates these feature sets into a single framework,

enabling the extraction of comprehensive structural, statistical, and empirical properties from SAT instances. SATfeatPy's design facilitates efficient and reproducible feature extraction, supporting tasks such as satisfiability prediction, problem categorisation, and model counting.

Key advantages of SATfeatPy include:

- **Unified Framework**: Combines multiple feature extraction methodologies, reducing the complexity of preprocessing SAT instances for ML tasks.
- **Extensibilit**y: Offers modular implementations, allowing researchers to extend the library with additional features or adapt it to new SAT domains.
- **Ease of Use**: Provides a straightforward API for extracting features from SAT instances, streamlining experimentation and analysis.

By leveraging SATfeatPy, this research integrates the rich feature sets of SATzilla, Ansótegui, and Alfonso into machine learning pipelines, enhancing the analysis and prediction capabilities for various SAT-related tasks. The library's public availability supports its adoption in broader SAT and ML research, fostering innovation in feature-based solver optimization.

3.3.5 Computational Overhead and Benefits of Feature Extraction

When employing machine learning models for SAT instance prediction tasks, feature extraction represents a necessary preprocessing step that introduces additional computational overhead. Although this step increases the total solution time, studies such as [XHHLB12, HXHLB14] have demonstrated that the strategic use of these extracted features to select the most appropriate solver or heuristic significantly outweighs the initial cost, resulting in an overall reduction in solving time. This improvement is particularly notable in algorithm portfolio approaches, where feature extraction costs are amortized across multiple solver invocations [BKK+16].

3.4 Method

3.4.1 Algorithms

This section provides a detailed description of the machine learning algorithms employed in this research, particularly focusing on Random Forest and XGBoost. These algorithms were chosen for their robustness, scalability, and their proven effectiveness in handling complex, high-dimensional data. Additionally, we include standard algorithms used for comparison in the model counting section.

3.4.1.1 Machine Learning

Machine learning algorithms are at the core of predictive modeling, offering powerful tools to extract patterns from data and make informed predictions. Among these, ensemble methods such as Random Forest and boosting algorithms like XGBoost have gained prominence due to their ability to handle large datasets, manage overfitting, and achieve high accuracy.

3.4.1.2 Random Forest

Random Forest is an ensemble learning method primarily used for classification and regression tasks [Bre01]. It operates by constructing multiple decision trees during training and outputting the mode of the classes for classification or the mean prediction for regression. The key advantages of Random Forest include its ability to handle large datasets with higher dimensionality, its resistance to overfitting, and its capability to manage missing values effectively.

A Random Forest is built using the following steps:

1. Bootstrap Sampling: Multiple subsets are created from the original dataset using sampling with replacement.
2. Decision Trees Construction: For each subset, a decision tree is constructed by selecting a random subset of features at each split, rather than considering all features. This introduces diversity among the trees.
3. Voting/Averaging: For classification tasks, the mode of the classes predicted by individual trees is taken as the final prediction. For regression tasks, the mean prediction of all trees is used.

One of the fundamental strengths of Random Forest is its ability to reduce variance through averaging, which leads to improved accuracy and robustness compared to individual decision trees. Additionally, the importance of features can be estimated based on how much they improve the splitting criterion (e.g., Gini impurity) across the forest [Bre01].

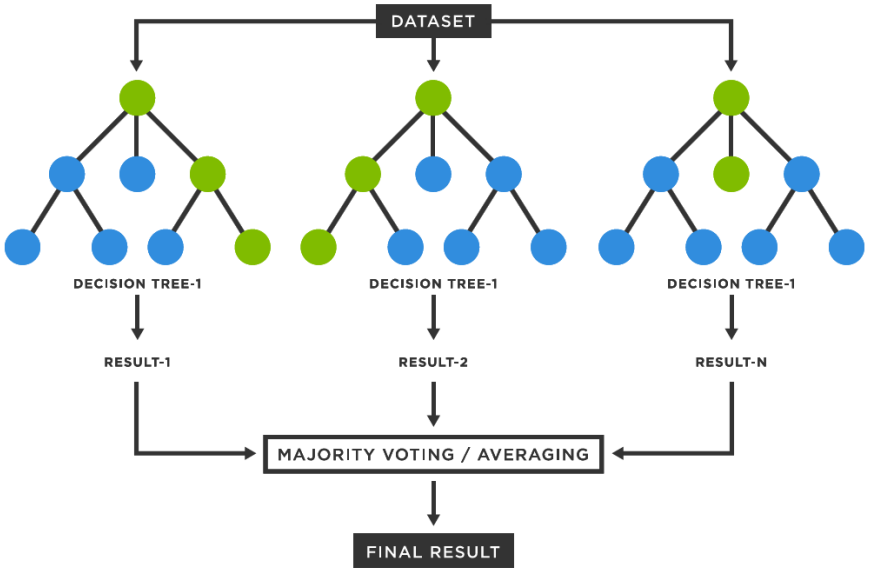


Figure 3.15: Random Forest Scheme [Tse21]

3.4.1.3 XGBoost

XGBoost (Extreme Gradient Boosting) is a scalable and efficient implementation of gradient boosting framework, widely used for supervised learning tasks such as classification and regression [CG16]. It builds an ensemble of decision trees in a sequential manner, where each tree attempts to correct the errors of the previous one. This method enhances performance and accuracy, making it a powerful tool for machine learning competitions and practical applications.

The key components of XGBoost include:

1. Gradient Boosting: Trees are added sequentially, and each new tree focuses on reducing the errors (residuals) made by the previous trees. This is achieved by optimizing a differentiable loss function.

2. Regularization: XGBoost includes regularization terms in the objective function to penalize model complexity, which helps prevent overfitting and improves generalization.

3. Tree Pruning: Unlike traditional boosting, XGBoost performs a more sophisticated tree pruning process to find the optimal structure of each tree.
4. Sparsity Awareness: XGBoost handles sparse data efficiently by employing a sparsity-aware split finding algorithm, making it suitable for high-dimensional and sparse datasets.

XGBoost's efficiency and scalability are attributed to its system design, which includes block structure for parallel computation, cache awareness for optimal hardware utilization, and out-of-core computing for handling large datasets that do not fit into memory [CG16].

3.4.1.4 Model Counters

Modern exact model counter devised a set of different heuristics and techniques to deal with relatively complex formulas. Some of those include caching the results of solved subsets of problems [ML98], dynamically dividing the remaining formulas into sub-problems [JP00] and caching their counts [BDP03] and combining the caching of components with standard conflict-driven clause learning (CDCL) to prune the search [SBB+04, Thu06]. Another extremely viable approach is *knowledge compilation*, where the original propositional formula is converted into a different representation in which the model count can be more efficiently computed [Dar01, HD07, MMBH12]. While these techniques have been shown to be extremely successful, the computational memory usage and runtime of such algorithms scale extremely poorly with the size of the problem, limiting the application of these to relatively small formulas. In order to deal with larger instances, new methods for obtaining an approximate evaluation of the number of solutions have been developed. These new model counters can be categorized into three different groups:

- (ϵ,δ) counters,
- lower (or upper) bounding counters,
- guarantee-less counters.

The counters belonging to the first category compute the probability that the calculated number of solutions will be in a certain interval, as described by the following equation: *[Correction:*

$$P\left(\frac{S}{1+\epsilon} \leq c \leq S(1+\epsilon)\right) \geq 1-\delta \tag{3.2}$$

] with S the actual number of solution, c the approximated number of solutions and $\epsilon, \delta \in [0, 1]$ the guarantees . To this category belong [KLM89] and [CMV16].
The second category provides a number of solution that is higher (or lower) than c with a probability of $1-\delta$. Examples are [GSS06] and [KSS08]. The final category features counters that do not provide any guarantees that can be very efficient and deliver practical good approximations [EGS12], [WS05]. These algorithms use different sampling techniques in order to reduce the amount of space searched and therefore scale much better than their exact counterpart. The method to calculate the number of favourable assignments to propositional formulas described in this paper does not feature counting algorithms with different heuristics but rather a learning approach.

Approximate Model Counter. *ApproxMC* [CMV16] is a hashing-based algorithm for approximate discrete integration over finite domains and provides (ϵ,δ) guarantees. The probability that the approximate count is within a certain distance of the true count is evaluated by the following Equation 3.2.

In other words, it provides a count of solutions with a certain probability of correctness within a given threshold. For our experiments, we set the configurations of guarantees to $\epsilon = 0.8$ and $\delta = 0.2$, i.e. the default configuration used by the authors as well as $\delta = 1$. This second parameter configuration effectively transforms *ApproxMC* into a guarantee-less model counter.

Probabilistic Exact Model Counter. *GANAK* [SRSM19] stands out as a state-of-the-art tool in the field of probabilistic exact model counting, harnessing the power of universal hash functions and a novel probabilistic component caching system. Unlike traditional model counters that may rely on deterministic methods, *GANAK* introduces a degree of probabilistic reasoning into the exact counting process, allowing it to provide counts with a specified level of confidence. This is made possible by the delta parameter, δ, which in the context of *GANAK*, dictates the confidence with which the model count is asserted to be exact. The innovative use of such probabilistic techniques enables *GANAK* to tackle complex counting problems with remarkable efficiency and scalability. For our experimental setup, *GANAK* was configured to run with two distinct delta values: $\delta = 0.2$ and $\delta = 1$.

3.4.2 Metrics

This section describes the metrics used for the assessment of the performances of different algorithms. The objectives of the experiments are threefold: SAT/UNSAT prediction, instance category classification and model counting regression. These are performed using machine learning algorithms based on features extracted from each instance. An analysis of the computational effort of the machine learning learning algorithm coupled with the feature extraction and the traditional algorithms is also performed. To ensure the robustness and generalizability of the machine learning models, we employed 10-fold cross-validation in all our experiments. Each model was trained and evaluated across ten different subsets of the data, where in each run, one subset was used for validation while the remaining nine were used for training.

For each instance, we extract the following features:

- **SATzilla base**: The base features from SATzilla [XHHLB08]. 38 features.
- **SATzilla probing**: SATzilla probing features. 31 features.
- **ANT**: Features from Ansotegui et al. [ABGCL17]. 4 features.
- **ALF**: Features from Alfonso et al. [AM14]. 254 features.

For SAT/UNSAT prediction and instance classification, performance is compared using different feature sets. The feature sets evaluated include SATzilla base, SATzilla full (base + probing), ANT (Ansotegui), ALF (Alfonso), and a combination of all features. Performance is assessed using the Random Forest Classifier (RFC) and XGBoost (XGB) algorithms. Additionally, the computational effort required to extract these features is compared. The metric chosen for this experiment is accuracy. This is a metric that measures the proportion of correct predictions made by the model out of the total number of predictions.

For model counting regression, we extracted all of the features and then compared the machine learning algorithms trained on those features with the standard state-of-the-art model counters. The following regression metrics have been used to evaluate the performance:

- Mean Absolute Error (MAE): MAE measures the average magnitude of the errors in a set of predictions, without considering their direction. It is the average over the test sample of the absolute differences between prediction and actual observation where all individual differences have equal weight.

- Root Mean Squared Error (RMSE): RMSE represents the root of the average of the squares of the errors, giving a larger weight to larger errors. It is calculated by taking the square root of the mean of the squared differences between the predicted and actual values. This metric is useful for highlighting significant errors.
- Coefficient of Determination (R^2): The R^2 metric, also known as the coefficient of determination, measures the proportion of the variance in the dependent variable that is predictable from the independent variables. It provides an indication of the goodness of fit of a model, with a value ranging from 0 to 1. An R^2 value close to 1 indicates that the model explains a large portion of the variance, while a value close to 0 indicates that the model does not explain much of the variance.
- Mean Absolute Percentage Error (MAPE): MAPE measures the accuracy of a model as a percentage by calculating the mean of the absolute percentage errors between the predicted and actual values. It is widely used to assess the performance of regression models, especially when dealing with data where the values are in different scales. MAPE is defined as the average of the absolute errors divided by the actual observed values, expressed as a percentage. However, it can be sensitive to small values in the actual data.

Finally, we investigate the statistical features that most affect the learning algorithms' decisions. Understanding their relevance in the estimation process can lead to interesting insights into which factors mostly affect the number of solutions of an SAT instance. This can be useful for ignoring the non-relevant features in the extraction process, speeding up the computational time. We use SHAP (SHapley Additive exPlanations) [LL17] to analyse the results and increase their explainability. SHAP is a popular and powerful method for interpreting machine learning models. It provides a way to explain the predictions of a model by assigning an importance value to each feature in the input data. These values represent the contribution of each feature to the final prediction made by the model. SHAP analysis calculates these importance values using a game-theoretic approach called Shapley values, which ensures that the values are fair and additive across all features. By using SHAP analysis, we can gain a better understanding of how a model works and in which direction the factors are driving its predictions.

3.5 Evaluation

This section presents the results of the experiments conducted to evaluate the performance of the machine learning algorithm in predicting satisfiability (SAT/UNSAT), problem category classification and model counting. All the experiments have been performed using a single core on an Intel(R) Xeon(R) Gold 6240 CPU 2.60GHz.

3.5.1 Feature Time analysis

The scalability of the feature extraction techniques is evaluated by recording the computational time required to generate the features for a dataset of randomly-generated CNFs. The dataset comprises 300 instances with a consistent clauses-to-variables ratio of 4.2 and uniformly increased size. Figure 3.16 shows the time required to compute each feature set as the problem size increases.

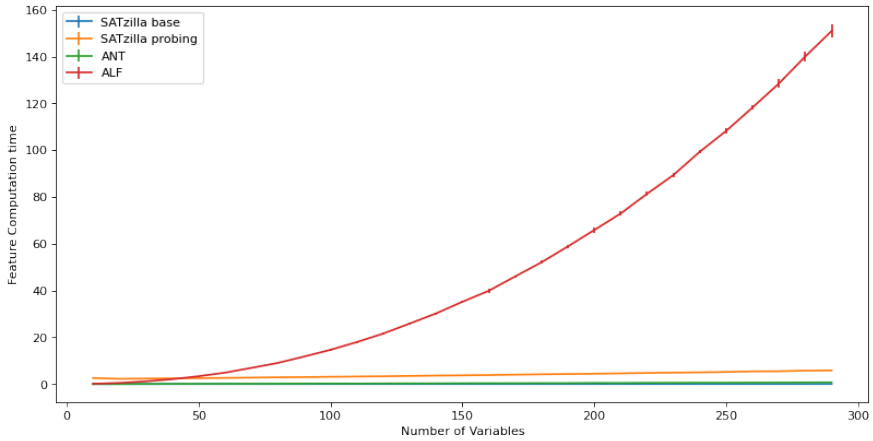


Figure 3.16: Time to generate features vs. size. The mean time to solve each set of instances is plotted in a line, and the variance is shown in the error bars.

The ALF feature set exhibits polynomial growth in computational time with increasing problem size, making it less suitable for large instances. This significant growth is primarily attributed to the computational cost of generating graph representations such as the Gate Graphs (AND/BAND/EXO), Resolution Graph (RG), and Binary Implication Graph (BIG). These graphs require detailed pairwise analysis of literals and clauses, resulting in higher computational complexity as the problem size increases. Among these, the Resolution Graph is particularly heavy due to the need to compute all resolvable pairs of clauses, which grows combinatorially with the number of clauses.

In contrast, the SATzilla feature sets (base and full) show linear growth, with SATzilla base being the fastest due to its exclusion of probing features. The probing features, such as DPLL and local search probing, introduce additional computational overhead as they involve partial runs on a user-defined interval of time (2 seconds in our case) of solving algorithms to gather empirical data. While these features provide valuable insights into the problem's empirical hardness, they are time-intensive, particularly for large-scale instances. The choice of feature set thus represents a trade-off between computational efficiency and the richness of the extracted information.

3.5.2 SAT/UNSAT Classification

The SAT/UNSAT classification task aims to determine whether a given CNF instance is satisfiable (SAT) or unsatisfiable (UNSAT). Table 3.3 shows the classification accuracies and the average computational times for each feature set across ten runs of 10-fold cross-validation.

Table 3.3: SAT/UNSAT classification accuracies and average computational times (in seconds) across feature sets.

Feature set	RFC Accuracy	XGB Accuracy	Comp. Time (s)
SATzilla base	91.21%	92.34%	0.03
SATzilla full	**99.47**%	**99.54**%	3.18
ANT	86.14%	86.23%	0.15
ALF	92.04%	94.73%	119.54
All	99.11%	99.52%	122.89

The SATzilla full feature set achieves the highest accuracy for both RFC and XGB, with accuracies of 99.47% and 99.54%, respectively. The difference between SATzilla full and the complete feature set (All) is minimal, as confirmed by Bayesian statistical tests [BCDZ17], which indicate that the two feature sets are practically equivalent. This result highlights the effectiveness of the SATzilla full set in preserving the most relevant information for the SAT/UNSAT classification task.

To further understand the contribution of individual features, we performed a SHAP (SHapley Additive exPlanations) analysis on the RFC model trained over the All dataset. The SHAP values, shown in Figure 3.17, reveal that the eight most important features all belong to the SATzilla full set, with several

probing features dominating. Among these, the gsat_BestSolution_Mean and saps_BestSolution_Mean capture the mean objective function value of the best solutions found by the GSAT [SLM92] and SAPS [HTH02] local search algorithms, respectively, providing insights into the empirical hardness of instances. The high SHAP values of these features indicate their critical role in predicting satisfiability, as they summarize key information about the quality of local search results. Similarly, features such as gsat_BestSolution_CoeffVariance and saps_BestSolution_CoeffVariance measure the variability of the best solutions' objective values, shedding light on the ruggedness of the search space and helping to distinguish between satisfiable and unsatisfiable instances.

Additionally, the feature estimate_log_number_nodes_over_vars, which estimates the logarithm of the number of search tree nodes divided by the number of variables, acts as a measure for the expected search complexity of the DPLL algorithm. This feature complements the local search probing metrics by providing an analytical perspective on problem difficulty. Features like rg_weights_min, derived from the Resolution Graph (RG), capture the minimum edge weight, reflecting the presence of easily resolvable clauses that simplify the problem structure. The importance of these features demonstrates the value of combining structural insights with empirical probing metrics.

The SHAP analysis confirms that probing features derived from local search algorithms are indispensable for effective SAT/UNSAT prediction. These features capture critical aspects of empirical hardness and search landscape characteristics, providing an edge over purely structural features such as those in the ANT and ALF sets. While these probing features come with additional computational overhead, as reflected in the 3.18 seconds required to compute SATzilla full compared to 0.03 seconds for SATzilla base, the significant improvement in accuracy justifies their inclusion. This trade-off between computational cost and predictive performance underscores the importance of these features in high-accuracy models.

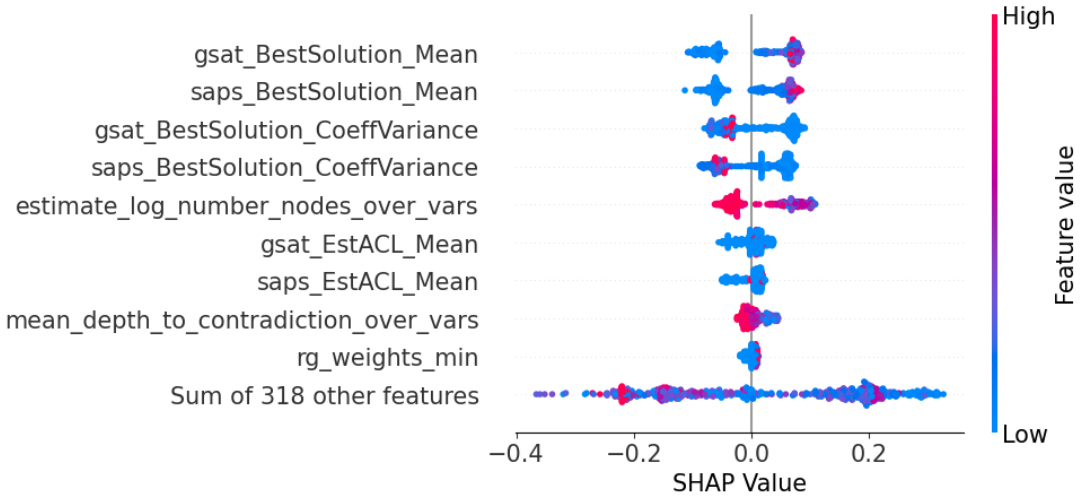


Figure 3.17: SHAP analysis for the RFC on the SAT/UNSAT classification task. The eight most important features, primarily derived from local search probing, highlight the importance of empirical hardness metrics.

3.5.3 Problem Category Classification

The problem category classification task aims to determine the category of a given CNF instance. Table 3.4 shows the classification accuracies and computational times for each feature set. The All set performed the best, with 99.81% and 99.79% accuracy for RFC and XGB, respectively. This high accuracy suggests that the combined feature sets effectively capture the structural characteristics of different problem categories.

Table 3.4: Problem category classification accuracies and computational times across feature sets for both Random Forest and XGBoost classifiers.

Feature set	RFC Accuracy	XGB Accuracy	Comp. Time (s)
SATzilla base	99.64%	99.63%	0.03
SATzilla full	99.66%	99.67%	3.18
ANT	93.04%	93.11%	0.15
ALF	99.77%	99.66%	119.54
All	**99.81%**	**99.79%**	122.89

The results reveal that while probing features play a critical role in SAT/UNSAT classification, graph-based features are more relevant for problem category classification. The SHAP analysis, shown in Figure 3.18, highlights the dominance of ALF features, particularly those derived from the Binary Implication

Graph (BIG) and AND-gate graph. Features like big_node_max, big_node_mean, and and_node_val_rate capture structural relationships between literals and clauses, which are more indicative of problem category than satisfiability.

The Binary Implication Graph (BIG) encodes implications between literals as directed edges, with features like big_node_max and big_node_mean reflecting the degree statistics of nodes in this graph. Higher values for these features often correspond to problem categories with densely connected variable relationships, such as 'php' (pigeonhole problems) or 'tseitin' encodings. Similarly, the AND-gate graph features, such as and_node_val_rate and and_weights_std, provide insights into the distribution and variability of logical AND connections within the formula, which are characteristic of categories like 'kcolor' and 'cliquecoloring'.

Another important feature is the binary_ratio, which measures the ratio of binary clauses to the total number of clauses. This feature is particularly relevant for categories with a high prevalence of binary clauses, such as 'php' and 'tseitin'. Additionally, rg_weights_mean, derived from the Resolution Graph, highlights the average resolvability of clauses, further distinguishing between categories based on their structural complexity.

The importance of graph-based features over probing features in this task is not surprising, as the problem category is inherently tied to the structural patterns within the CNF instance. Probing features, while effective for capturing empirical hardness, do not provide as much information about the categorical structure of a problem. This shift in feature relevance also explains the superior performance of the ALF set for this task, despite its higher computational cost. The graph-based representations in ALF, such as the BIG and AND-gate graph, offer a nuanced characterization of the instance's structure, making them indispensable for problem category classification.

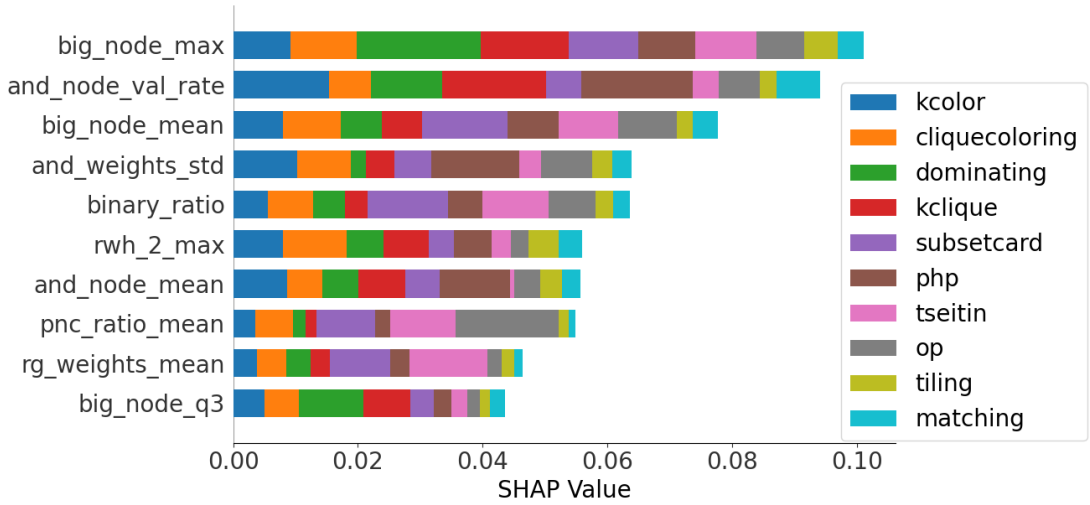


Figure 3.18: SHAP analysis for the RFC on the problem category classification task. The most relevant features are derived from graph-based representations, such as the Binary Implication Graph and AND-gate graph, highlighting their importance for capturing structural characteristics.

3.5.4 Solver Selection Based on Instance Type

Given the variety of SAT instance types identified in the multi-class classification, selecting the most suitable solver for each type can significantly enhance solving efficiency. For graph-based problems such as *kcolor*, *cliquecoloring*, *dominating*, and *kclique*, specialized solvers such as `Glucose` [AS12] typically demonstrate strong performance due to their efficient conflict-driven clause learning (CDCL) heuristics. For combinational or arithmetic-based instances such as *subsetcard*, *php*, and *op*, solvers like `MapleSAT` [LGPC16] and `Kissat` [FH20] have consistently outperformed others, given their optimized heuristics for structured combinatorial constraints. Additionally, problem types involving structured combinatorial constraints, such as *matching*, *tiling*, and *tseitin*, often benefit from solvers tailored to highly structured instances, notably `Lingeling` [FH20], which leverage preprocessing techniques to simplify complex structural relationships.

3.5.5 Model Counting

This section empirically compares the machine learning algorithms to a state-of-the-art approximate model counter and probabilistic exact model counter. We evaluate them in terms of regression accuracy and computational effort required to calculate the prediction. Using the trained learning models, we

investigate the statistical features that mainly affect the number of solutions and the possibility of approximating the problem with a close formula. The machine learning algorithm are trained on statistical features extracted from the instances of the training sets and evaluated on the test sets for Dataset 1, Dataset 2 and Dataset 4. For Dataset 3, a 10-fold cross-validation for the RFR has been performed on the total number of instances. This decision is due to the large diversity of instances in this set.

3.5.5.1 Accuracy of the Approximation

Tables 3.5, 3.6 3.7 and 3.8 show the performance of the learning models versus the approximate model counter.

GANAK delivers the best overall performance in the three datasets. In Dataset 1 and Dataset 2 the probabilistic exact model counter provides the same exact number of solutions irrespective of the parameter δ. It is worth noting though that for the first set GANAK was not able to reach a solution within 5000 seconds for 21 instances in case $\delta = 0.2$ and 15 for $\delta = 1$. The machine learning methods show a MAE, RMSE and MAPE of an order-of-magnitude greater than the state-of-the-art approximate model counter. Both ApproxMC and the learning algorithms perform better on Dataset 1. We observe that by relaxing the confidence parameter in the approximate model counter the result worses by a 2-3 factor. The learning algorithms have metrics that are worse by factors between 2-10, especially the RMSE and the MAPE, which tend to magnify instances with larger errors. This behaviour is highlighted in Figures 3.19 and 3.20. Here, the plots show the distribution for the two testing datasets of the GANAK model count, with a grey interval representing the targeted guarantees for ApproxMC, the approximation of ApproxMC, and the prediction of the RFR. We can see that while not as accurate as the approximate model counter, the learning algorithms provide an estimate that is within the boundaries for most of the solutions. In fact, if we evaluate the Equation (3.2) using the adopted guarantee of $\epsilon = 0.8$, we obtain a probability for the predictions of the RFR to be inside the guarantee interval of respectively 0.91 for Dataset 1 and 0.85 for Dataset 2, which is consistent with the one expected for a $\delta = 0.2$.

Dataset 3 incorporates a large number of difficult model counting instances stemming from a wide range of applications. We observe here that GANAK still provides overall the best results. ApproxMC with guarantees actually has a better value of MAE, and very similar performance in terms of RMSE. The RFR

and XGB reach very close performances with the guarantee-less ApproxMC, providing a better value of RMSE. Most importantly, the learning algorithm is able to output a number of solutions for all of the 1597 instances, while the timeout of 5000 seconds is reached for over 204 and 201 instances for, respectively, approximately with or without guarantees and 476 for GANAK. The distribution of the model count values for the different techniques is shown in 3.21.

Dataset 4 shows that for the difficult instances of the Model Counting Competitions, the differences in performance between the learning algorithm and the approximate model counter are magnified. It is worth noting that the very limited number of available instances for training negatively affected machine learning techniques. Moreover, it is worth noting that the Random Forest Regressor was able to provide a solution for all 30 of the testing instances, while ApproxMC and Ganak did not manage to compute the model count for 12 problems. The distributions of solutions for Dataset3 are shown in 3.22. The instances for which ApproxMC was not able to provide a solution are plotted as 0.

Table 3.5: Accuracy comparison for Dataset 1. For this table, only 429 instances out of 450 are considered, as those are the only ones where ApproxMC and GANAK are able to provide a count within the 5000 seconds timeout.

Method	**MAE**	**RMSE**	**R2**	**MAPE**
ApproxMC ($\delta = 0.2$, $\epsilon = 0.8$)	0.020	0.025	0.999	0.002
ApproxMC ($\delta = 1$)	0.054	0.068	0.999	0.006
GANAK ($\delta = 0.2, 1$)	7.988×10^{-9}	1.037×10^{-8}	1	7.801×10^{-10}
RFR	0.097	0.208	0.996	0.020
XGB	0.099	0.209	0.998	0.021

Table 3.6: Accuracy comparison for Dataset 2.

Method	**MAE**	**RMSE**	**R2**	**MAPE**
ApproxMC ($\delta = 0.2$, $\epsilon = 0.8$)	0.034	0.046	0.999	0.003
ApproxMC ($\delta = 1$)	0.054	0.068	0.999	0.006
GANAK ($\delta = 0.2, 1$)	1.853×10^{-11}	1.208×10^{-10}	1	5.376×10^{-12}
RFR	0.125	0.173	0.991	0.043
XGB	0.121	0.168	0.990	0.044

Table 3.7: Accuracy comparison for Dataset 3. For this table, only 993 instances out of 1597 are considered, as those are the only ones where ApproxMC and GANAK are able to provide a count within the 5000 seconds timeout.

Method	**MAE**	**RMSE**	**R2**	**MAPE**
ApproxMC ($\delta = 0.2$, $\epsilon = 0.8$)	0.772	6.622	0.998	0.027
ApproxMC ($\delta = 1$)	1.937	23.876	0.979	0.043
GANAK ($\delta = 0.2, 1$)	1.568	5.682	0.998	0.005
RFR	2.884	23.357	0.980	0.134
XGB	2.765	22.245	0.98	0.128

Table 3.8: Evaluation metrics comparison for Dataset 4, including the 12instances where ApproxMC and GANAK did not provide a model count within a 5000 seconds timeout (Dataset 3 Extra).

	Dataset 4 Test				Dataset 4 Extra			
Method	**MAE**	**RMSE**	**R2**	**MAPE**	**MAE**	**RMSE**	**R2**	**MAPE**
ApproxMC ($\delta = 0.2$, $\epsilon = 0.8$)	0.005	0.013	0.999	0.0004	-	-	-	-
ApproxMC ($\delta = 1$)	0.184	0.116	0.995	0.060	-	-	-	-
Ganak ($\delta = 0.2, 1$)	0.0001	0.005	1	0.0001	-	-	-	-
RFR	2.201	3.382	0.929	0.286	1.163	2.028	0.996	0.218
XGB	2.246	3.291	0.933	0.281	1.168	2.044	0.996	0.235

Figures 3.19, 3.20, and 3.21 illustrate the distributions of model count values for GANAK, ApproxMC, and RFR across the testing datasets.

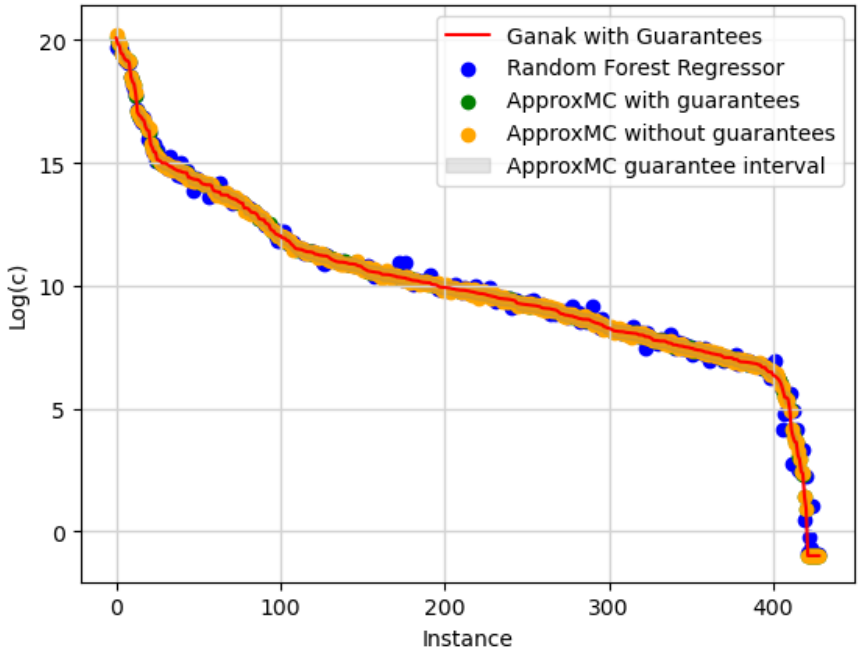


Figure 3.19: Plot of GANAK solutions, ApproxMC approximate solutions ($\delta = 0.2, 1$), ApproxMC guarantee interval, and Random Forest Regression predicted solutions for Dataset 1 testing instances. The decimal logarithm of the number of solutions (Y-axis) for each instance (X-axis) is plotted in descending order.

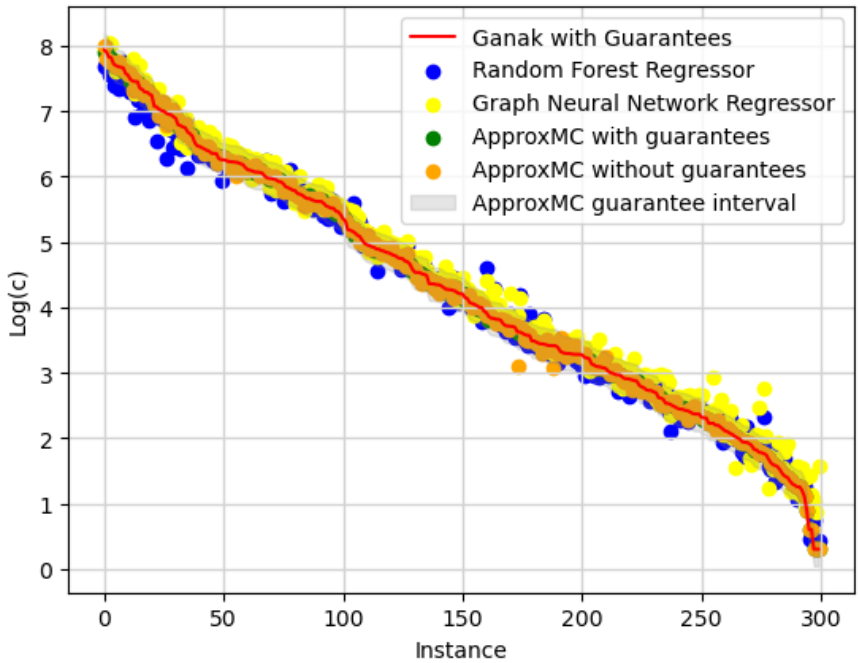


Figure 3.20: Plot of GANAK solutions, ApproxMC approximate solutions ($\delta = 0.2, 1$), ApproxMC guarantee interval, and Random Forest Regression predicted solutions for Dataset 2 testing instances. The decimal logarithm of the number of solutions (Y-axis) for each instance (X-axis) is plotted in descending order.

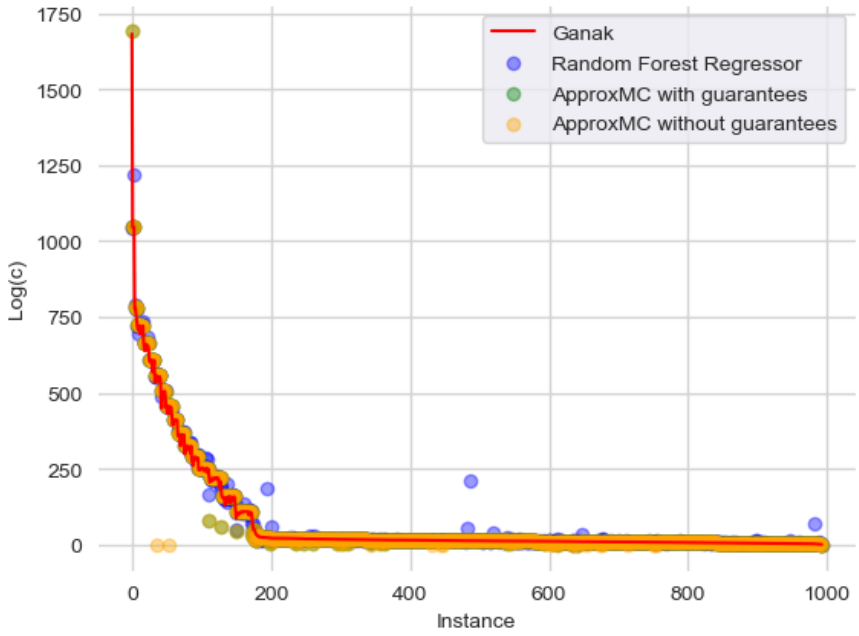


Figure 3.21: Plot of GANAK solutions, ApproxMC approximate solutions ($\delta = 0.2, 1$), and Random Forest Regression predicted solutions for Dataset 3 testing instances. The decimal logarithm of the number of solutions (Y-axis) for each instance (X-axis) is plotted in descending order.

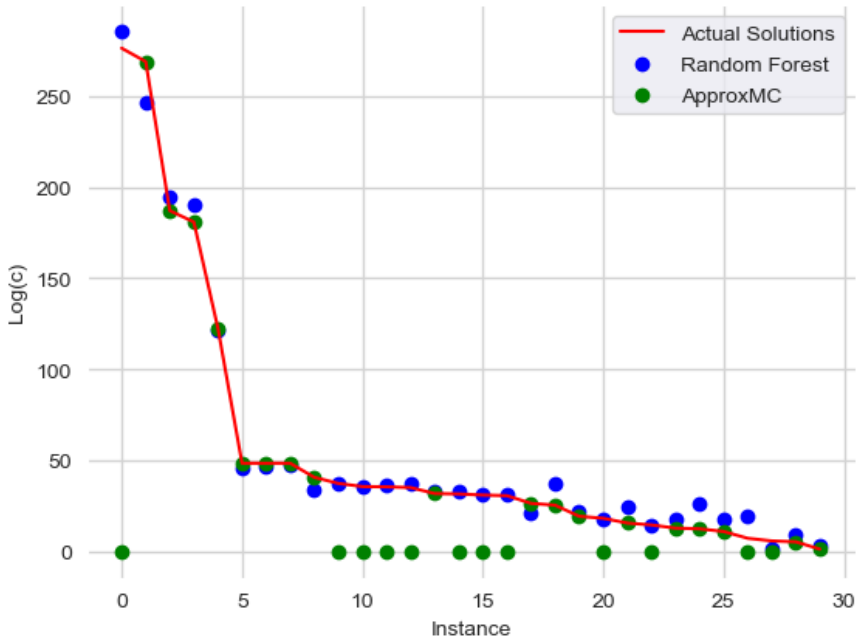


Figure 3.22: Plot of GANAK solutions, ApproxMC approximate solutions ($\delta = 0.2, 1$), and Random Forest Regression predicted solutions for Dataset 4 testing instances. The decimal logarithm of the number of solutions (Y-axis) for each instance (X-axis) is plotted in descending order.

3.5.5.2 Computational Time

Tables 3.9, 3.10 present the computational times in seconds required to produce the results of the previous section. ApproxMC and GANAK do not require a training phase and can be directly deployed on the test instances. For the machine learning approach, the steps needed to predict the number of solutions of a CNF are feature extraction of the training set, model fitting, feature extraction of target CNF, and model prediction. The strength of learning algorithms is their ability to provide almost instantaneous prediction after a training time. This characteristic makes them very interesting when computational power and time are limited. In our case, most of the time is needed to extract the features from the instances; the tool used [PDVO22] is a prototype implemented in *Python* [VRD09]. We observe that a part from Dataset 2, where the size of the instances is very small and the algorithmic approach is extremely quick, the feature extraction and training time is always much more efficient. Moreover, while model counters do not scale well and require a considerable amount of computing time and power for larger instances, the feature extraction is extremely consistent and scales much more efficiently with the problem size.

Finally, for both Dataset 1, Dataset 3 and Dataset 4, both model counters reached the 5000 seconds timeout on multiple instances, thus failing to provide an answer, while our approach can make a prediction in less than one millisecond.

Table 3.9: Comparison of computational times for Dataset 1 and Dataset 2. For Dataset 1, ApproxMC ($\delta = 0.2$) has a maximum runtime of 3324 seconds, with GANAK reaching up to 4958 seconds. Dataset 2 results are significantly faster, with ApproxMC and GANAK completing within less than a second. Times are reported for both training and testing phases across different methods.

Method	**Dataset 1**				**Dataset 2**			
	Train		**Test**		**Train**		**Test**	
	Mean	**Max**	**Mean**	**Max**	**Mean**	**Max**	**Mean**	**Max**
$AMC_{\delta=0.2}$	-	-	38	3324	-	-	0.05	0.31
$AMC_{\delta=1}$	-	-	12.80	152.59	-	-	0.013	0.04
$Ganak_{\delta=0.2}$	-	-	677	4823	-	-	0.071	0.43
$Ganak_{\delta=1}$	-	-	674	4958	-	-	0.069	0.26
Feat. Ex.	2.37	2.98	2.27	2.86	2.31	2.69	2.17	2.61
RFR	355.98		10^{-4}		478.12		10^{-4}	
XGB	399.02		10^{-4}		501.7		10^{-4}	

Table 3.10: Comparison of computational times for Dataset 3 and Dataset 4. For Dataset 3, ApproxMC ($\delta = 0.2$) reaches the timeout of 5000 seconds in over 204 instances, and GANAK in 201 instances. For Dataset 4, counters reach the timeout for 12 instances using ApproxMC and GANAK. Times are reported for training and testing phases across all methods.

Method	**Dataset 3**				**Dataset 4**			
	Train		**Test**		**Train**		**Test**	
	Mean	**Max**	**Mean**	**Max**	**Mean**	**Max**	**Mean**	**Max**
$AMC_{\delta=0.2}$	-	-	5.72	1172	-	-	1190	4967
$AMC_{\delta=1}$	-	-	3.82	689	-	-	584	1257
$Ganak_{\delta=0.2}$	-	-	45.30	3860	-	-	3830	4998
$Ganak_{\delta=1}$	-	-	45.08	3881	-	-	3866	4991
Feat. Ex.	2.37	2.98	2.23	2.89	2.23	2.89	2.27	2.63
RFR	521.3		10^{-4}		117.3		10^{-4}	
XGB	566.3		10^{-4}		134.3		10^{-4}	

3.5.5.3 Feature Importance

As we did before, we investigate the statistical features that mostly affect the regressors' decisions. Understanding their relevance in the estimation process can lead to interesting insights into which factors mostly affect the number of solutions of a SAT instance. This information can be used to engineer additional features specifically relevant for predicting the model count and to provide new statistical features. Figures 3.23, 3.24, 3.25, 3.26 show the SHAP analysis results for the Random Forest Regressor on the three datasets.

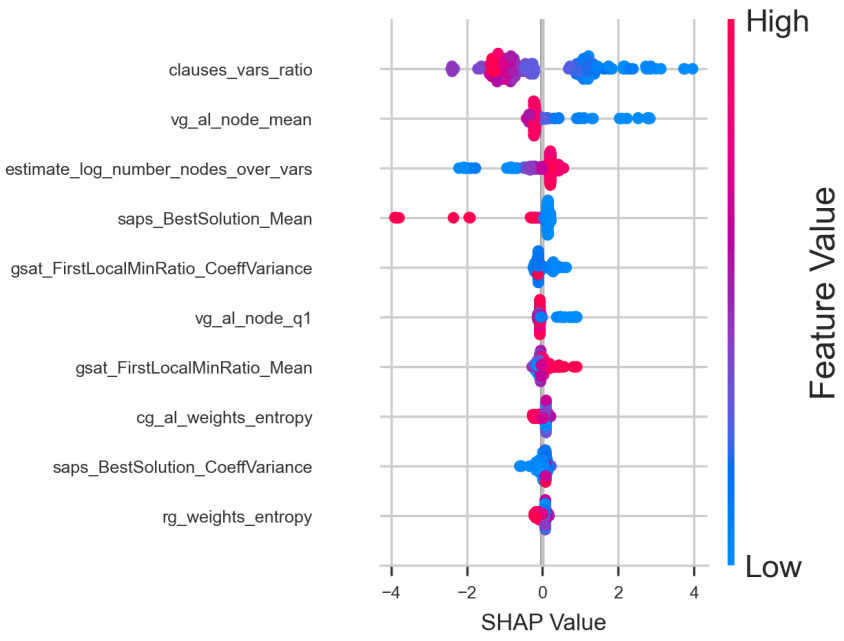


Figure 3.23: SHAP analysis performed for the Random Forest Regressor on Dataset 1 testing set.

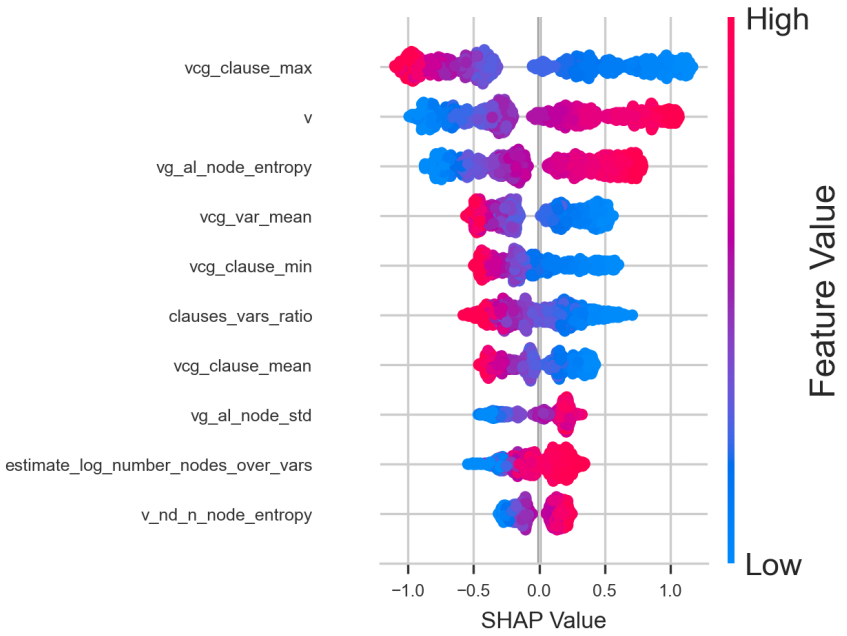


Figure 3.24: SHAP analysis performed for the Random Forest Regressor on Dataset 2 testing set.

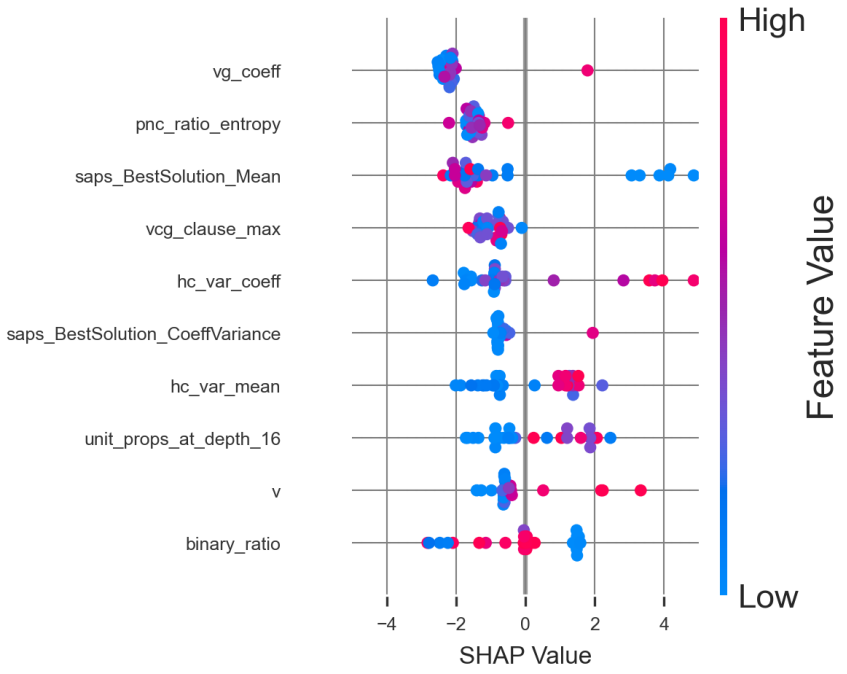


Figure 3.25: SHAP analysis performed for the Random Forest Regressor on Dataset 3.

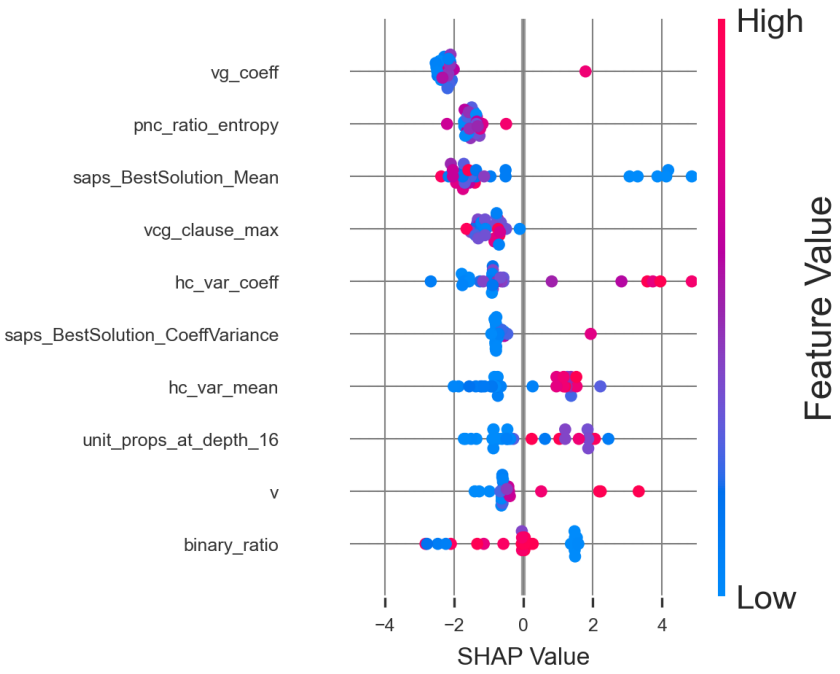


Figure 3.26: SHAP analysis performed for the Random Forest Regressor on Dataset 4.

In Dataset 1, the most relevant feature is the average number of clauses in which a variable appears; a higher value of this ratio leads to a smaller number of solutions. This is intuitive since being involved in fewer constraints allows the variables to have multiple assignments that do not make the formula unsatisfiable, and so increases the number of solutions. The second most important feature belongs to the set stemming from the variable graph of the instance. The third feature, estimate_log_number_nodes_over_vars, is of particular interest. This feature belongs to the DPLL probing feature set. It is an estimate of the number of nodes in the search tree, which can be regarded as an approximate measure of the search space.

Other features that have a considerable impact on the total number of models are gSat BestAvgImprovement_Mean, gSat_FirstLocalMinStep_Mean, saps_BestSolution_Mean and saps_BestSolution_CoeffVariance. These are obtained by probing the search space with the stochastic local search algorithms GSAT and SAPS.

The algorithms are run multiple times until the search trajectory reaches a plateau that cannot be escaped within a given number of steps. It seems that measurements of statistics and estimations of the search space have a significant contribution to the calculation of the total number of solutions.

For Dataset 2, the most relevant features belong to the ones extracted respectively from the Variable Clause Graph and Variable Graph representation of the instances. These encapsulate the structure of the instance, while v (the number of variables) is associated with its size. Unsurprisingly, instances with more variables have a higher number of solutions. It is worth noting as well that estimate log number nodes over vars and clauses vars ratio provide important contributions to this dataset as well.

For Dataset 3 and Dataset 4, the most important features belong to both graph sets and probing sets. Especially for these datasets, which incorporate a large number of instances with different structures and characteristic, it is shown that both a graph representation of the CNF, as well as an accurate estimation of the search size space are extremely important to provide a correct prediction of the number of solutions.

In the analysis, we can see that the features selected are dataset dependent. However, information relative to the number of variables and clauses, and estimations of the search space size are fundamental regardless of the instance

type. This insight suggests that improving features estimating the search space would lead to a more accurate approximate model counting.

3.6 Model Count Guarantees

ApproxMC employs a hashing-based partitioning technique to probabilistically estimate the number of satisfying assignments of a Boolean formula. Utilizing universal hash functions, it partitions the solution space into manageable subsets, counts the solutions within these partitions, and aggregates the results to provide an overall estimate. The methodology offers probabilistic guarantees on the accuracy of the estimate, ensuring that the derived count is within a specified confidence interval of the true solution count with high probability.

In contrast, machine learning algorithms, due to their inherent reliance on empirical data and non-deterministic optimization processes, cannot provide analogous exact probabilistic guarantees on their predictions or classifications. However, we are currently exploring the potential of providing probability distributions for machine learning regressors in the context of model counting. The aim is to bridge the gap between deterministic model counting tools like ApproxMC and the probabilistic nature of machine learning predictions, offering a more nuanced understanding of the solution landscape.

3.7 Conclusions

This chapter presented an exploration of machine learning techniques for predicting satisfiability, problem categories, and model counts of SAT instances based on extracted features. The results demonstrate the effectiveness of various feature sets in accurately classifying and predicting SAT properties, highlighting the robustness of models like Random Forest and XGBoost in this domain. Notably, the integration of structural, statistical, and probing features significantly enhances prediction accuracy, particularly when combining feature sets from different sources.

The analysis also emphasizes the trade-off between feature extraction time and prediction accuracy, underscoring the importance of selecting appropriate features based on the specific task at hand. This is particularly useful in real-world scenarios where SAT solvers are deployed, such as hardware verification, AI planning, and combinatorial optimization. By selecting features that are com-

putationally efficient yet informative, machine learning models can be tailored to suit the constraints of specific applications, such as real-time decision-making systems or large-scale industrial optimization tasks. For example, in domains like circuit design or logistics planning, where SAT solving is used iteratively or embedded in larger workflows, reducing feature extraction overhead can directly improve system performance and scalability.

SHAP analysis provided valuable insights into the most influential features. These insights are especially critical for designing interpretable and efficient models, as they help prioritize features that contribute most to predictive performance. Additionally, understanding the relevance of different features across tasks—such as the dominance of probing features for SAT/UNSAT classification versus graph-based features for problem category classification—can inform task-specific feature engineering strategies, further enhancing the utility of machine learning in SAT research.

The investigation into model counting using machine learning algorithms revealed that while traditional exact and approximate counters like ApproxMC and GANAK provide strong guarantees and accuracy, machine learning models offer competitive performance with much faster prediction times. This positions machine learning as a valuable tool, particularly in scenarios where computational resources or time are limited. For instance, in probabilistic reasoning or uncertainty quantification applications, where model counts are used as inputs to higher-level algorithms, the ability to quickly approximate counts using machine learning can enable near-real-time computations. This flexibility extends the applicability of model counting to domains where traditional approaches may be too slow, such as large-scale probabilistic graphical models or time-sensitive planning tasks.

Chapter 4

Deep Learning Predictions

Abstract. *This chapter explores the application of deep learning (DL) to Boolean Satisfiability (SAT) and its effectiveness across three key tasks: satisfiability prediction, instance classification, and model counting. By leveraging diverse SAT instance encodings—including binary, image-based, and graph-based representations—deep learning models such as Convolutional Neural Networks (CNNs), Graph Neural Networks (GNNs), and Variational Autoencoders (VAEs) are benchmarked against traditional machine learning (ML) approaches. Our analysis highlights the ability of DL models to overcome limitations of handcrafted features, automatically learning meaningful representations from raw or minimally processed data. This chapter is based on the work of our previous publication in* ***ICTAI: Automated SAT Problem Feature Extraction Using Convolutional Autoencoders*** [DVO21] and the code is available. [1]

4.1 Introduction

The application of deep learning (DL) to SAT has gained significant momentum in recent years, driven by the ability of these models to learn complex patterns and representations from raw data automatically [SLB+18]. Unlike traditional machine learning (ML) methods, which rely heavily on hand-engineered features extracted from SAT instances, deep learning approaches can directly process various encodings of SAT instances to perform various tasks [LSL+22].

[1] https://github.com/mardalla/SmartSAT

This capability to operate on raw or minimally processed data represents a significant advancement, particularly in a domain where feature extraction is non-trivial and often requires domain-specific knowledge. Traditional feature extraction is a strongly lossy process that focuses on human-defined features, many of which are highly correlated between each other or might not be relevant for the decision-making process itself. DL and these encodings can overcome these limitations by automatically learning relevant features from the data. In this context, we discuss the two following approaches: automatic feature extraction and direct application of DL models.

In this chapter, we explore the effectiveness of deep learning models across several key tasks in SAT research: satisfiability prediction, instance classification, and model counting. We apply various deep learning models to these tasks, including Convolutional Neural Networks (CNNs), Convolutional Autoencoders (CAEs), Graph Neural Networks (GNNs), and Graph Variational Autoencoders (GVAEs). Each model is tested with distinct SAT instance encodings to assess their performance. These tasks are crucial for enhancing the performance of SAT solvers, optimizing solver selection, and improving our understanding of the structural properties of SAT instances.

The choice of encoding is a critical factor in the performance of deep learning models. In our study, we utilize a diverse range of encodings, such as binary encodings, image-based encodings, and graph-based representations. Each of these encodings highlights different aspects of the SAT instance structure, demonstrating the versatility of deep learning models.

The integration of deep learning into Boolean Satisfiability has seen considerable progress, beginning with early foundational studies and gradually advancing to more complex applications in problem solving and generation. The pioneering work by Bünz et al. [BL17] in exploring GNNs for classifying SAT problems laid the groundwork for this integration. Their approach demonstrated that GNNs could learn features of satisfiability in a weakly-supervised setting, opening the door to further explorations in the field.

Building upon these early explorations, Wang et al. [WR18] leveraged the use of deep reinforcement learning for learning SAT solver heuristics. This marked a significant advancement in applying deep learning to symbolic reasoning, showcasing the potential for more intuitive and efficient heuristic development in SAT solving. The introduction of NeuroSAT by Selsam et al. [SLB+18] further exemplified the adaptability of neural networks in solving SAT problems,

demonstrating their ability to generalize and tackle a diverse range of problems beyond their training scope. The integration of GNNs with *Stochastic Local Search (SLS)* SAT solvers by Zhang et al. [ZSZ+20] and Kramer et al. [KB23] significantly enhanced solver efficiency, refining the initialization process and overall effectiveness of SLS solvers. This integration not only improved performance but also highlighted the potential of GNNs in understanding and solving complex SAT instances. In specialized domains, the application of deep learning was further explored by Sun et al. [SGBP20], who evaluated NeuroSAT in SAT problems for cryptanalysis. Their findings revealed that specialized solvers like the one they propose, NeuroGIFT, could significantly outperform general-purpose solvers or standard algorithmic approaches in highly structured problems, indicating the potential for tailored neural network solutions in specific problem domains. Chang et al. [CZL22] introduced a framework using Graph Attention Networks for predicting the satisfiability of domain-specific SAT problems, demonstrating notable improvements in predictive accuracy, particularly in random 3-SAT problems. The learning of SAT solver heuristics through deep reinforcement learning was carried out further by Yolcu et al. [YP19]. Their approach showcased the ability of learned heuristics to efficiently find satisfying assignments, thereby reducing the computational steps required. Finally, the application of GNNs in reasoning 2-Quantified Boolean Formulas (2QBF) was investigated by Wang et al. [WYC+19], providing insights into the challenges and potential of applying machine learning tools to more complex symbolic reasoning problems. Chen et al. [CBZ+19] introduced Deep Reasoning Networks (DRNets), combining deep learning with reasoning for solving complex tasks, including standard combinatorial problems like SAT. This approach exemplified the potential of deep learning in enhancing logical reasoning and problem-solving capabilities. This chapter presents a comprehensive comparison of deep learning approaches across these tasks. We benchmark these approaches against traditional ML methods that rely on handcrafted features, evaluate the models using various performance metrics, and discuss the trade-offs between different deep learning architectures and encodings. This thorough comparison provides a clear understanding of the relative strengths and weaknesses of each approach.

4.2 Encoding Techniques for SAT Problem Representation in Deep Learning Models

In the application of deep learning to the SAT, the choice of encoding is crucial, as it directly impacts the effectiveness of the neural networks in learning meaningful representations of the SAT instances. The encoding transforms the SAT problem, typically expressed in CNF, into a format suitable for processing by deep learning models. This section provides an in-depth discussion of three prominent encoding techniques: Binary Encoding [DVO21], Image-based Encoding [LMSS16], and Graph-based Encoding [BL17], each of which has distinct advantages and is suitable for different types of neural network architectures.

4.2.1 Binary Encoding

This approach differs from traditional methods of representing SAT instances as binary matrices by employing a more detailed and structured conversion process aimed at preserving the logical relationships within the SAT problem as well as the handling using convolutional layers.

Encoding Process

1. **Representation of Clauses and Variables**:
    - Consider a SAT instance with C clauses and V variables.
    - Each clause i in the SAT instance is represented by an array of binary values, where each variable j is encoded into two adjacent binary positions in the array.
    - Specifically:
        - The first position, $x_{i,2j}$, is set to 1 if the variable j appears positively in the clause i, otherwise it is set to 0.
        - The second position, $x_{i,2j+1}$, is set to 1 if the variable j appears negatively in the clause i, otherwise it is set to 0.
        - If a variable does not appear in the clause, both positions are set to 0.

- This results in a binary array of size $2V$ for each clause, effectively doubling the number of variables in the original SAT instance.

2. **Formation of the Binary Matrix**:
   - The binary arrays representing all clauses in the SAT instance are combined into a binary matrix of dimensions $C \times 2V$.
   - Each row in this matrix corresponds to a clause, while each pair of adjacent columns corresponds to the presence (positive or negative) of a specific variable across the clauses.
3. **Symmetry Breaking**:
   - We implement a symmetry-breaking process to reduce the impact of invariances in the SAT problem. This involves:
     - **Variable Negation Symmetry**: Ensuring that for each variable, the normal literal appears at least as frequently as the negated one.
     - **Variable Permutation Symmetry**: Ordering variables based on their frequency of appearance in the formula, with ties broken by the frequency of their negated literals.
     - **Clause Permutation Symmetry**: Ordering clauses by their cardinality (number of literals), with ties broken by the index of the first literal.
4. **Example**:
   - Consider the SAT instance:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$$

   - The binary encoding would proceed as follows:

x_1	$\neg x_1$	x_2	$\neg x_2$	x_3	$\neg x_3$
1	0	0	1	0	0
0	1	0	0	1	0
0	0	1	0	0	1

   - Here, the first row represents the clause $(x_1 \vee \neg x_2)$, where x_1 is set

to 1 for x_1, $\neg x_2$ is set to 1 for $\neg x_2$, and the other positions are set to 0 as those variables are absent in this clause.

Advantages of Binary Encoding

- **Precision**: This encoding preserves the exact structure of the SAT instance, allowing the deep learning model to leverage the full complexity of the logical relations within the formula.
- **Scalability**: The approach is scalable to instances with varying numbers of variables and clauses, with the symmetry-breaking steps helping to reduce computational redundancy.
- **Compatibility**: This encoding is particularly well-suited for use in convolutional networks, which excel at identifying patterns and relationships in grid-like data structures.

4.2.2 Image-based Encoding

The image-based encoding leverages the visual representation of SAT instances, transforming the problem into an image-like structure. This approach was notably explored by Loreggia et al., where the SAT instance is represented as a grayscale image through a process that involves ASCII-based conversion [LMSS16]. The transformation process involves the following steps:

1. **ASCII Conversion**: Each SAT instance, expressed in CNF, is first converted into a textual representation using ASCII encoding. The CNF formula is written out as a sequence of characters, where each variable and clause is represented by its corresponding ASCII character codes. This conversion turns the CNF formula into a stream of integer values corresponding to the ASCII codes.
2. **Grayscale Mapping**: The ASCII values obtained from the previous step are then mapped to grayscale intensities to create an image. Each ASCII code is interpreted as a pixel intensity in the range [0, 255]. This results in a grayscale image where the textual structure of the SAT instance is preserved as patterns of varying intensities.
3. **Image Structure**: The ASCII-encoded values are arranged in a 2D grid to form an image. The dimensions of this image depend on the length of the CNF representation and can be adjusted (e.g., via padding or reshaping)

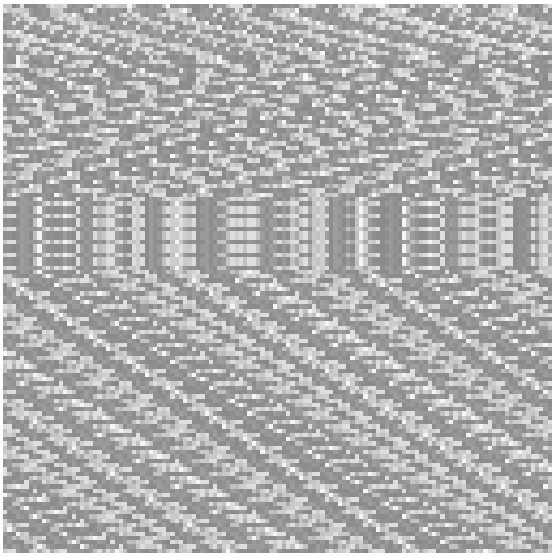


Figure 4.1: Image extracted from SAT problem instance.

to maintain uniform image dimensions across different instances. The resulting image encapsulates the logical structure of the SAT problem in a form that is suitable for processing by Convolutional Neural Networks (CNNs).

In this way, the image reflects the structural and syntactic features of the SAT instance, enabling CNNs to recognize patterns that correlate with satisfiability or other instance properties. This method preserves much of the the logical and syntactical structure of the SAT problem in a format that is readily processable by deep learning models designed for image recognition tasks.

CNNs are particularly adept at processing these images due to their ability to detect spatial hierarchies and patterns. The network can learn to identify patterns that correlate with the satisfiability of the instance or other characteristics such as the problem's complexity or the type of SAT instance.

However, the limitations of this encoding include potential loss of relational information between variables that are not spatially close in the image representation. Additionally, the image size can grow large for instances with many variables and clauses, requiring careful management of input dimensions.

Figure 4.1 show an image extracted from a SAT instance.

4.2.3 Graph-based Encoding

Graph-based encoding represents SAT instances as graphs, allowing deep learning models to capture the relational structure between variables and clauses effectively. This section explores several graph-based encoding techniques, each providing a unique perspective on the problem's structure.

1. **Variable-Clause Graph (VCG)**:
    - **Nodes**: Two types of nodes are used—one set representing variables and another representing clauses.
    - **Edges**: An edge is drawn between a variable node and a clause node if the variable appears in the clause. The edges can be labeled to indicate whether the variable is present as a positive or negative literal.
    - **Example**: For the SAT instance $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$, the VCG would include nodes for x_1, x_2, x_3, and for each of the three clauses. Edges would connect x_1 to the first clause node, x_2 to the first and third clause nodes, and so on, with edge labels indicating positive or negative literals.
2. **Literal Incidence Graph (LIG)**:
    - **Nodes**: Each literal in the SAT instance (both positive and negative) is represented as a node.
    - **Edges**: An edge is drawn between two literal nodes if they both appear in the same clause. This type of graph emphasizes the co-occurrence relationships between literals.
    - **Example**: Using the same SAT instance, the LIG would have nodes for x_1, $\neg x_2$, $\neg x_1$, x_3, x_2, and $\neg x_3$, with edges connecting literals that share a clause, such as x_1 and $\neg x_2$ being connected due to their co-occurrence in the first clause.
3. **Variable Incidence Graph (VIG)**:
    - **Nodes**: Only variables are represented as nodes.
    - **Edges**: An edge is drawn between two variable nodes if they both appear in the same clause. This representation highlights the relationships between variables based on their co-occurrences in clauses.
    - **Example**: The VIG for our SAT instance would include nodes for x_1,

x_2, and x_3 with edges between x_1 and x_2, x_1 and x_3, and x_2 and x_3 based on their shared clauses.

4. **Literal-Clause Graph (LCG)**:
    - **Nodes**: This is a bipartite graph where nodes represent either literals or clauses.
    - **Edges**: An edge exists between a literal node and a clause node if the literal is part of the clause. This graph explicitly captures the relationship between literals and the clauses they belong to.
    - **Example**: In the LCG for the SAT instance, nodes would include literals x_1, $\neg x_2$, $\neg x_1$, x_3, x_2, $\neg x_3$, and clause nodes for each clause. Edges would connect each literal to its corresponding clause node.

These graph representations are particularly well-suited for Graph Neural Networks (GNNs), which process data structured as graphs [KPKT24]. GNNs can perform message passing, where nodes iteratively update their embeddings based on information from their neighbours, allowing the network to learn rich representations that capture both local and global structures within the SAT instance.

The primary advantage of graph-based encoding is its ability to retain and exploit the relational structure of the SAT problem, making it ideal for tasks that require a deep understanding of variable interactions and clause dependencies. However, the complexity of constructing and processing these graphs can be high, especially for large SAT instances, requiring efficient graph processing techniques and careful selection of graph features.

For this chapter, only the VCG has been used to train deep learning algorithms. In the next chapter all of the described graphs will be used for evaluation. The VCG has been used in the following works [BL17, CZL22]. The main reasons are its ability to capture the structural relationships between variables and clauses, enabling efficient feature extraction for GNNs, and its role in reducing symmetries, which helps to minimize redundant search paths and improve learning focus within the neural network model.

4.2.4 Efficiency of Deep Learning Encodings

Deep learning models bypass manual feature extraction by automatically computing internal encodings from SAT instances. Although encoding computation

introduces minor overhead, it is substantially faster than traditional feature extraction, especially when leveraging GPU acceleration. Recent research indicates that these learned encodings enable rapid solver selection and heuristic tuning, leading to tangible improvements in solver performance [SLB+18, ZCC+24]. Thus, the slight computational cost of encoding computation is justified by the resulting efficiency gains in the solving process.

4.3 Method

This section outlines the methodology employed for training and testing the deep learning algorithms used for predicting satisfiability, categorizing instances, and model counting.

4.3.1 Training Approach

The models were designed for three core tasks:

- **Binary Classification (SAT/UNSAT)**: Determine whether a SAT instance is satisfiable.
- **Multi-Class Classification (Instance Category)**: Classify instances into predefined categories.
- **Regression (Model Counting)**: Predict the number of satisfying assignments.

The training framework includes supervised models (CNN-BE, CNN-IE, and GNN) and unsupervised models (CAE and GVAE). Direct approaches handle classification and regression tasks, while feature extraction models provide compressed representations for downstream analysis.

4.3.2 Data Preparation

4.3.2.1 Dataset Description

The datasets include SAT problems described in Sections 3.2.1 and 3.2.2, encoded in the DIMACS format. Each instance is labelled with satisfiability, category, and model count. Metadata in leading comments specifies instance properties.

4.3.2.2 Preprocessing

Instances are encoded into binary matrices, image and graph representations for neural network processing. Symmetry-breaking preprocessing is applied to binary-encoded data. For graph-based models, instances are represented as Variable-Clause Graphs (VCGs). Datasets are split into training, validation, and test sets to ensure balanced evaluation.

4.3.3 Model Architectures

4.3.3.1 Overview of Models

The models employed include:

- ***CNN-BE***: Convolutional Neural Network that processes binary-encoded SAT instances with convolutional layers capturing clause and literal-level patterns.
- ***CNN-IE***: Convolutional Neural Network that operates on image-encoded SAT instances, learning spatial relationships through convolutional layers.
- ***GNN***: Graph Neural Network that leverages VCG representations to model variable-clause dependencies.
- ***CAE***: Convolutional Autoencoder, i.e. unsupervised model for feature extraction and instance reconstruction that uses the binary encoding.
- ***GVAE***: Graph Variational Autoencoder, i.e. unsupervised model for feature extraction and instance reconstruction that uses the VCG representation.

4.3.3.2 Direct Approaches

CNN-BE. This CNN model using binary encoding processes SAT instances represented as binary matrices. This model is structured to identify and learn patterns within these binary-encoded SAT instances for multiple tasks.

The architecture is as follows:

- **Input Layer**: The input is a binary matrix of dimensions $C \times 2V$, representing the SAT instance.
- **First Convolutional Layer**: A single filter of size 2×1 is applied with no overlap. This layer captures individual literals across all clauses. The op-

eration of a standard convolutional layer can be mathematically described at the matrix level as:

$$Y^{(k+1)} = \sigma\left(X^{(k)}W^{(k)} + B^{(k)}\right)$$

where: X is the input matrix (feature map) of size $C \times 2V$, W is the convolutional filter matrix of size $M \times N$, B is the bias matrix added to the result of the convolution,Y is the output feature map matrix, andσ is the activation function applied element-wise to the resulting matrix (e.g. RELU).

- **Second Convolutional Layer**: 128 filters with a window size of 500×1 are applied without overlap, covering all literals within each clause. This layer detects clause-level patterns.
- **Dropout and Batch Normalization**: A dropout layer with a rate of 0.1 is followed by batch normalization to prevent overfitting and stabilize the training process.
- **Third Convolutional Layer**: 256 filters with a window size of 5×1 are applied to learn more complex patterns.
- **Fully Connected Layers**: Two fully connected layers follow the convolutional layers. The first layer has 64 neurons and serves to integrate the features learned by the convolutional layers, while the second layer with 32 neurons reduces the dimensionality further.
- **Output Layers**:
  - **Binary Classification (SAT/UNSAT)**: A fully connected layer with 1 neurons and a sigmoid activation function is used to produce a probability distribution over the SAT/UNSAT classes.
  - **Category Classification (instance category)**: A fully connected layer with N_c neurons (where N_c is the number of instance categories) and a softmax activation function to classify the SAT instance into one of the predefined categories.
  - **Regression (Model Counting)**: A fully connected layer with 1 neuron and a linear activation function is used to predict the model count as a continuous value.

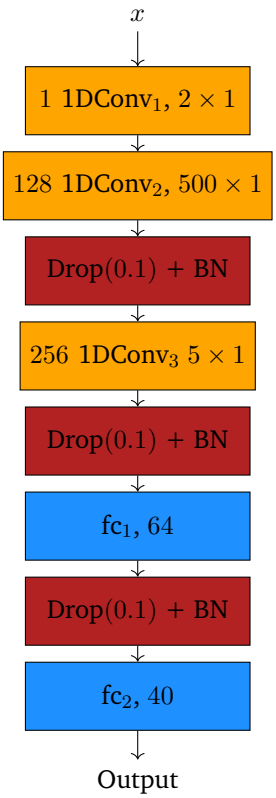


Figure 4.2: Convolutional Neural Network architecture.

CNN-IE. This CNN model using image encoding processes SAT instances represented as grayscale images. This model is designed to identify and learn spatial patterns within the image-encoded SAT instances for multiple tasks.

The architecture includes the following components:

- **Input Layer**: The input is a grayscale image of a fixed size (e.g., 128x128 pixels) representing the SAT instance.
- **First Convolutional Layer**: 32 filters of size 3×3 are applied with ReLU activation. This layer captures local patterns in the image.
- **Max-Pooling Layer**: A max-pooling layer with a 2×2 window is used to reduce the spatial dimensions of the feature maps.
- **Dropout Layer**: A dropout layer with a rate of 0.25 is applied to prevent overfitting.
- **Second Convolutional Layer**: 64 filters of size 3×3 are applied with ReLU activation. This layer captures more complex patterns.
- **Max-Pooling and Dropout Layers**: Another set of max-pooling and dropout

layers follows to further downsample the feature maps and regularize the model.

- **Third Convolutional Layer**: 128 filters of size 3×3 are applied with ReLU activation. This layer captures even higher-level features.
- **Max-Pooling and Dropout Layers**: A final set of max-pooling and dropout layers ensures that the network focuses on the most critical features while maintaining robustness against overfitting.
- **Fully Connected Layers**: The network concludes with two fully connected layers. The first has 512 neurons and integrates the learned features into a compact representation. The second layer with 256 neurons further refines this representation.
- **Output Layers**: Identical to *CNN-BE*.

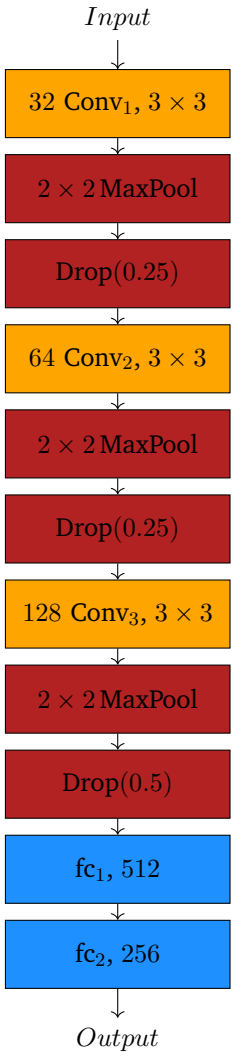


Figure 4.3: Convolutional Neural Network architecture for image-encoded SAT instances.

GNN. This GNN is designed to process SAT instances represented as VCGs. This encoding captures the relationships between variables and clauses in a bipartite graph structure. The GNN utilizes multiple layers of graph convolution to iteratively refine the node embeddings and capture the complex dependencies within the SAT instances. After that, the node embeddings are aggregated and passed through fully connected layers to perform classification and regression tasks, including SAT/UNSAT prediction, instance category classification, and model counting. The architecture is inspired by [BL17] and [SB19].

The architecture of the GNN is detailed below:

- **Input Layer**: The input is a VCG representing the SAT instance, where nodes correspond to variables and clauses, and edges represent the oc-

currence of variables in clauses.

- **Graph Convolution**:
  - The network begins with an initial node embedding for each variable and clause. These embeddings are iteratively updated using a message-passing process where information is exchanged between connected nodes.
  - Each graph convolutional layer applies 128 filters to propagate information across the graph's nodes, capturing local dependencies among variables and clauses. The operation is mathematically described by:
    $$H^{(k+1)} = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(k)}W^{(k)}\right)$$
    where $W^{(k)}$ are the learnable weights, $H^{(k)}$ is the node embedding matrix at the k-th layer, A is the adjacency matrix, and σ is the activation function (SELU).
  - Two layers of graph convolution refine the node embeddings, allowing the network to learn increasingly complex interactions between variables and clauses.
- **Activation and Normalization**: After each graph convolutional layer, SELU (Scaled Exponential Linear Unit) [KUMH17] activation is applied for non-linearity, followed by layer normalization to stabilize learning and improve convergence by reducing internal covariate shifts.
- **Pooling Layer**: After the message-passing layers, a mean aggregation is applied to pool the node embeddings into a single graph embedding, which represents the entire SAT instance in a condensed form.
- **Fully Connected Layers**: The graph-level embedding is passed through fully connected layers:
  - The first fully connected layer has 128 neurons and integrates the features learned by the graph convolutional layers.
  - The second fully connected layer has 64 neurons and further processes the embedding in preparation for the final task-specific outputs.
- **Output Layers**: Identical to *CNN-BE*.

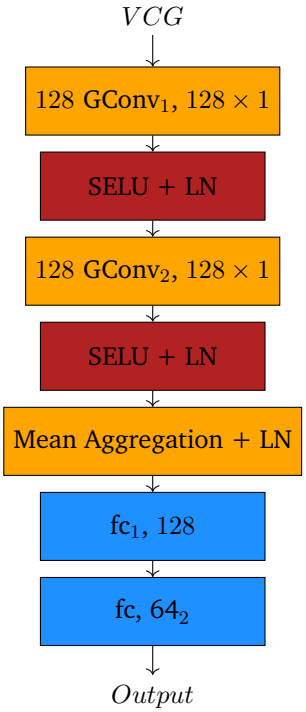


Figure 4.4: Graph Neural Network (GNN) architecture. The output layers are adapted for binary classification (SAT/UNSAT), category classification, and regression tasks.

4.3.3.3 Feature Extraction Methods

CAE. This CAE is designed to learn a compact representation of the input data, given as a binary encoding, through a process of encoding and subsequent decoding. This approach is particularly useful for reducing the dimensionality of data while preserving key structural features, making it highly suitable for tasks like SAT instance generation, where capturing the essence of complex problem structures is crucial.

The CAE consists of two main components:

- Encoder: The encoder maps the input data (e.g., a SAT problem instance) into a lower-dimensional latent space. It does this by applying convolutional layers that progressively reduce the spatial dimensions of the input while preserving its most essential features. This compressed representation is a learned abstraction of the input data.
- Decoder: The decoder attempts to reconstruct the original input from the latent representation. The convolutional layers in this stage expand the latent space back to the original dimensions, ideally reproducing the input

as closely as possible. The network minimizes the difference between the input and the reconstructed output through a loss function (often mean squared error).

The architecture of the CAE allows it to effectively capture hierarchical features from the data, which is particularly advantageous in SAT instance generation, where complex dependencies between variables must be encoded in a compact and meaningful way. The architecture includes the following components:

- **Input Layer**: The input is the same binary matrix used in the CNN with binary encoding.
- **Encoder**:
  - **First Convolutional Layer**: This layer applies a single filter of size 2×1 to the input matrix, capturing individual literals and transforming them into feature maps.
  - **Second Convolutional Layer**: With 128 filters and a window size of 500×1, this layer processes the entire set of literals within each clause, learning clause-level patterns.
  - **Dropout and Batch Normalization**: A dropout layer with a rate of 0.1 followed by batch normalization is applied to ensure that the network does not overfit and to stabilize the training process.
  - **Third Convolutional Layer**: This layer, with 256 filters and a window size of 5×1, captures more complex, high-level patterns within the SAT instance.
  - **Fully Connected Layers**: Two fully connected layers follow, with 64 and 32 neurons respectively. The final layer outputs the latent space representation z, which is a compressed version of the input data.
- **Decoder**:
  - **Fully Connected Layers**: The decoder mirrors the encoder's fully connected layers, expanding the latent space representation back into a higher-dimensional feature space.
  - **Transpose Convolutional Layers**: These layers reverse the convolutional operations performed by the encoder, gradually reconstructing the original input matrix from the latent space representation.

– **Output Layer**: The final layer reconstructs the binary matrix from the latent space, aiming to minimize the difference between the original input and the reconstructed output.

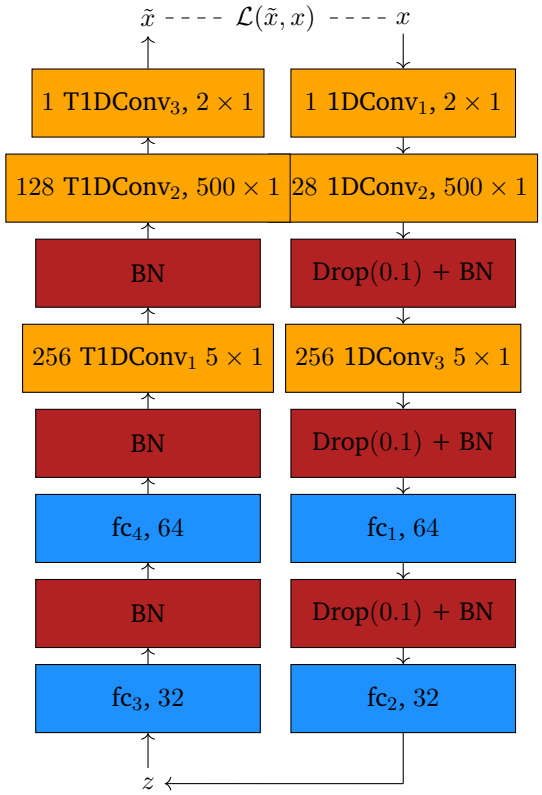


Figure 4.5: Convolutional Autoencoder architecture. The decoder mirrors the encoder structure with transposed convolutions, reconstructing the input from the latent space representation z.

GVAE. This GVAE is designed to perform unsupervised learning on a VCG representation of the data. It compresses the graph-encoded input into a lower-dimensional latent space and then reconstructs the input from this representation. The GVAE is composed of an encoder and a decoder, similar to the structure used in the Convolutional Autoencoder but adapted for graphs. The architecture of the GVAE is largely aligned with the standard structure for GVAEs, following the approaches described by [KW16] and [ZSNL22].

The GVAE architecture includes the following components:

- **Input Layer**: The input is a Variable-Clause Graph (VCG) representing the SAT instance, where nodes represent variables and clauses, and edges capture the relationships between them.
- **Encoder**:

  - **Graph Convolutional Layers**: The encoder begins with a series of graph convolutional layers that propagate information through the graph, capturing both local and global dependencies between nodes. These layers apply the graph convolution operation, which updates each node's embedding based on its neighbors' features.
  - **Activation and Normalization**: After each graph convolutional layer, SELU (Scaled Exponential Linear Unit) activation is applied for non-linearity, followed by layer normalization to stabilize learning and improve convergence by reducing internal covariate shifts.
  - **Pooling Layer**: A mean aggregation function is applied to pool the node embeddings into a single graph embedding, which summarizes the entire structure of the input graph.
  - **Fully Connected Layers**: Following the pooling layer, the graph embedding is passed through fully connected layers. These layers generate the mean and variance for the latent space distribution, which is then sampled to produce the latent representation z. This sampling step allows the model to learn a probabilistic latent space, facilitating the generation of diverse outputs.

- **Decoder**:
  - **Fully Connected Layers**: The decoder mirrors the encoder's fully connected layers, expanding the latent representation z back into a higher-dimensional space. These layers progressively reconstruct the graph structure from the latent code.
  - **Reconstruction Layer**: The final layer of the decoder reshapes the output to match the original graph's structure, using a hyperbolic tangent (tanh) activation function to produce outputs in the range $[-1, 1]$. This layer ensures that the reconstructed graph closely resembles the original input graph, minimizing the reconstruction loss.
- **Output Layer**: The output of the decoder is a reconstructed VCG, which is compared against the original graph to compute the reconstruction loss. The goal of the GVAE is to minimize this loss, thereby improving the quality of the learned latent space and the fidelity of the reconstructed graphs.

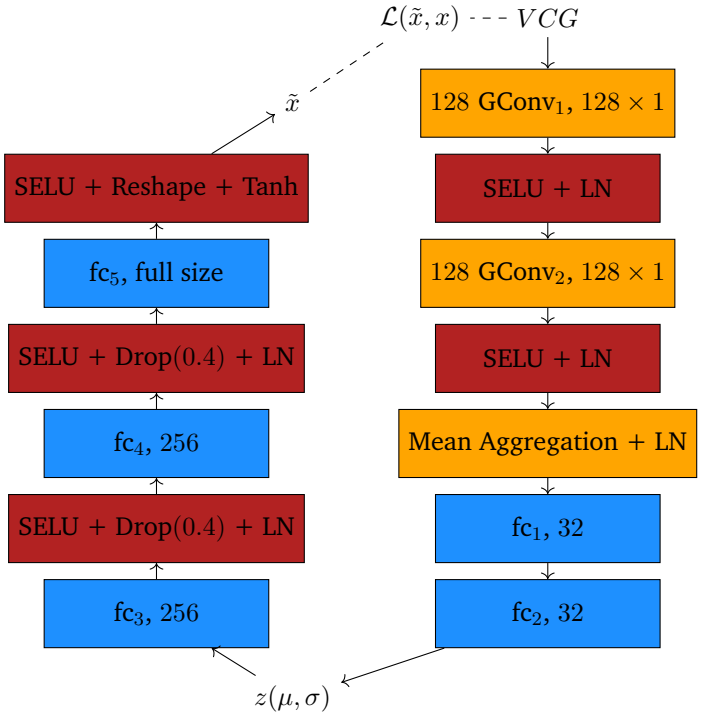


Figure 4.6: Graph Variational Autoencoder (GVAE) architecture.

4.3.4 Training and Hyperparametrization

The models were implemented and executed on NVIDIA Quadro RTX 8000 GPUs. The implementation utilized Keras [C+15] for deep learning models and Keras Spektral [GA20] for graph-based models. All the layers used are standard and implemented in the library.

During training, each dataset was split into training, validation, and test sets. For SAT/UNSAT and category prediction, a training and validation set was composed of 2700 instances, equally distributed between categories and SAT/UNSAT. We used an 80/20 split. The remaining 300 instances were used as testing. For model counting, the dataset of 1000 instances was used as training/validation, with an 80/20 split. A testing set of 300 instances is assembled.

Both CNN and GNN were trained to predict the satisfiability, the category, and the model count directly. The CAE and GVAE were trained only on reconstructing the SAT instances. Then, a Random Forest algorithm was trained on the latent space in order to predict satisfiability, category and model count.

4.3.4.1 CNN Models

The two CNN models were trained to perform classification and regression SAT instances based respectively on their binary and image-based encodings. For the CNN models, the primary hyperparameters included the number of convolutional layers, filter sizes, strides, and learning rates. The first CNN was designed with three convolutional layers, while the second utilized five layers to capture deeper features. The filter sizes were set to 3×3 and 5×5 with strides of 1. The learning rate was initialized at 0.001, and an Adam optimizer [KB17] was employed with a weight decay of 0.01 to prevent overfitting. The batch size was set to 16, as it offered a balance between training speed and model generalization.

The CNN models were employed for three types of tasks: binary classification, multi-class classification, and regression. The following loss functions were used:

- **SAT/UNSAT prediction**: For SAT/UNSAT prediction, which is a binary classification task, the **Binary Cross-Entropy** is utilized. This loss function is commonly used for binary classification problems where the output is expected to be a probability value between 0 and 1. The BCE loss is defined as:

$$\mathcal{L}_{BCE} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right)$$

  where y_i is the true label (0 or 1), $\hat{y}_i$ is the predicted probability, and N is the number of samples. This loss function penalizes incorrect classifications, especially when the predicted probability is far from the actual label, effectively guiding the model to learn the decision boundary between the two classes.

- **Category Prediction**: For category prediction, i.e. multi-class classification tasks, where the output is a categorical label among multiple classes, the **Cross-Entropy Loss** (also known as Softmax Loss) was used. This loss function is suitable for problems where the model needs to output a probability distribution over several classes. The Cross-Entropy Loss is defined as:

$$\mathcal{L}_{CE} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{p}_{ij})$$

where y_{ij} is a binary indicator (0 or 1) if class label j is the correct classification for instance i, $\hat{p}_{ij}$ is the predicted probability of instance i being in class j, and C is the total number of classes. This loss function encourages the model to increase the probability for the correct class while decreasing it for the incorrect ones.

- **Model Count Regression**: for the regression task, where the model needs to predict a continuous value rather than a categorical label, the **Root Mean Squared Error (RMSE)** was used as the loss function. RMSE is particularly effective for measuring the differences between values predicted by a model and the actual observed values. It is defined as:

$$\mathcal{L}_{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}$$

where $\hat{y}_i$ is the predicted value, y_i is the actual value, and N is the number of samples. RMSE is a good measure of how well a regression model is performing because it penalizes large errors more than smaller ones, providing a sensitive measure of prediction accuracy.

4.3.4.2 Convolutional Autoencoder (CAE)

The CAE model was trained to learn a compressed representation of SAT instances and reconstruct the original input from its latent representation. The training process aimed to minimize the binary cross-entropy loss, which treats each binary variable of the encoding as a binary classification task. This loss function was chosen due to the binary encoding of the input data.

The training was conducted using the Adam optimizer [KB17] to accelerate the back-propagation process. The initial learning rate was set to 10^{-4} with a decay rate of 10^{-5}, allowing for fine-tuning as the training progressed. To prevent overfitting and stabilize the training, dropout and batch-normalization layers were applied throughout the network. The dropout rate was set at 0.3 after each layer, except for the final layer, which did not use dropout. A batch size of 16 was chosen to accommodate the high memory demands of the CAE model during training.

The latent space dimension was explored with values ranging from $\delta = \{8, 16, 24, 32, 64\}$, with 32 selected based on cross-validation results to achieve a good trade-off between compression and reconstruction quality.

4.3.4.3 Graph Neural Network (GNN)

The GNN was trained to learn node and edge embeddings from variable-clause graph representations of SAT problems. Training involved supervised learning with categorical cross-entropy loss. At each epoch, the model aggregated information from neighboring nodes and updated embeddings. The learning rate was set to 0.0005, and the Adam optimizer [KB17] was used. The batch size was 16, limited by the computational intensity of processing graph data. L2 regularization with a weight decay of 0.0001 was applied to encourage generalization and prevent overfitting.

4.3.4.4 Graph Variational Autoencoder (GVAE)

The GVAE model extends the standard autoencoder framework to handle graph data. During training, the GVAE learned probabilistic embeddings in a latent space that were used to generate new SAT instances. The training process involved minimizing a composite loss function, combining the reconstruction loss for the input graph and the Kullback-Leibler (KL) divergence loss to maintain latent space continuity. The composite loss function $\mathcal{L}$ is defined as:

$$\mathcal{L} = \mathcal{L}_{\text{recon}} + \beta \cdot \mathcal{L}_{\text{KL}},$$

where $\mathcal{L}_{\text{recon}}$ is the reconstruction loss that measures how well the output graph $\hat{G}$ reconstructs the original input graph G, in this case the Binary Cross-Entropy. The KL divergence loss, $\mathcal{L}_{\text{KL}}$, is defined as:

$$\mathcal{L}_{\text{KL}} = D_{\text{KL}}(q(z|G) \| p(z)),$$

where $q(z|G)$ is the approximate posterior distribution of the latent variable z given the input graph G, and $p(z)$ is the prior distribution, typically a standard normal distribution. The parameter β controls the weight of the KL divergence term, balancing the trade-off between reconstruction accuracy and latent space regularization.

The GVAE utilized a two-layer graph convolutional encoder and a three-layer fully connected decoder. The latent dimension was set between 8 and 64, with 32 selected based on cross-validation results to achieve a good trade-off between compression and reconstruction quality. The learning rate was fixed at 0.0001, optimized using the Adam optimizer [KB17], which effectively manages adaptive learning rates in generative models. A batch size of 16 was chosen to manage memory consumption during training. Dropout rates of 0.3 in the encoder and L2 regularization with a weight decay factor of 0.0005 were applied to maintain model stability and generalization capacity.

4.4 Evaluation

This section presents our evaluation of the performances of the deep learning algorithms in terms of classification and regression accuracy for the tasks of satisfiability prediction, instance classification and model counting regression. The deep learning algorithms are compared then to the results obtained in chapter 3.7, using a Random Forest algorithm trained on statistical features extracted from the SAT instances.

4.4.1 SAT/UNSAT Prediction

The results presented in Table 4.1 show a comparative analysis of deep learning (DL) and machine learning (ML) algorithms for SAT/UNSAT prediction tasks. The traditional ML approach achieves the highest overall accuracy of 99.11%, with both SAT and UNSAT prediction accuracies reaching 99.8%. Among the DL models, the Graph Neural Network (GNN) demonstrates the closest performance to the ML approach, achieving an overall accuracy of 99.2%, which indicates that the graph-based representation of SAT problems effectively captures the structural relationships necessary for satisfiability prediction. The Binary CNN model follows with an accuracy of 97.3%, showing that binary encoding is more effective than image-based encoding, as the Image CNN only reaches 90.1% accuracy. This highlights the importance of selecting appropriate data representations for DL models. The Convolutional Autoencoder combined with a Random Forest Regressor (CAE + RFR) and the Graph Variational Autoencoder with a Random Forest Regressor (GVAE + RFR) also perform well, achieving accuracies of 97.0% and 97.7%, respectively. The slightly better performance of GVAE + RFR over CAE + RFR indicates that the graph-based la-

tent space learned by GVAE captures more relevant features for the SAT/UNSAT task than the convolutional autoencoder. These results demonstrate that while ML models maintain superior accuracy, DL models, particularly those utilizing graph-based techniques like GNN and GVAE, show strong potential for effective SAT problem prediction.

Table 4.1: SAT/UNSAT accuracy prediction for the DL algorithm compared with the ML approach

Algorithm	**Accuracy**	**% SAT**	**% UNSAT**
RFR (features)	99.1%	99.0%	**99.2**%
CNN-IE	90.1%	90.0%	90.2 %
CNN-BE	97.3%	97.6%	97.0 %
GNN	**99.2**%	**99.2**%	99.0%
CAE + RFR	97.0 %	96.6 %	97.4 %
GVAE + RFR	97.7 %	97.8 %	97.6 %

4.4.2 Comparison between automatically extracted features and handcrafted features

In this section we compare the features extracted using the deep learning algorithm with the handcrafted features of 3.3. We project all the instances of the dataset into increasing size latent spaces using the CAE and the GVAE, $\delta = \{8, 16, 24, 32, 40\}$. The goal is to test the ability of the approach to compress the instance structure. We train the autoencoders for the different δ values on the training set. For the handcrafted features, we compress them using principal component analysis (PCA) to reduce their cardinality. This is done to have a comparison between sets of same cardinality.

Figure 4.7 show the results of our experiment on the train and the test set, respectively. The orange dotted line is the accuracy achieved by using the full features set. The GVAE outperforms the CAE and the compressed handcrafted set, indicating that the deep learning algorithms are able to extract a bigger amount of information automatically for this particular task.

4.4.3 Instance Category Prediction

The results presented in Table 4.2 demonstrate the performance of various algorithms in classifying different types of SAT instances. The Random Forest Re-

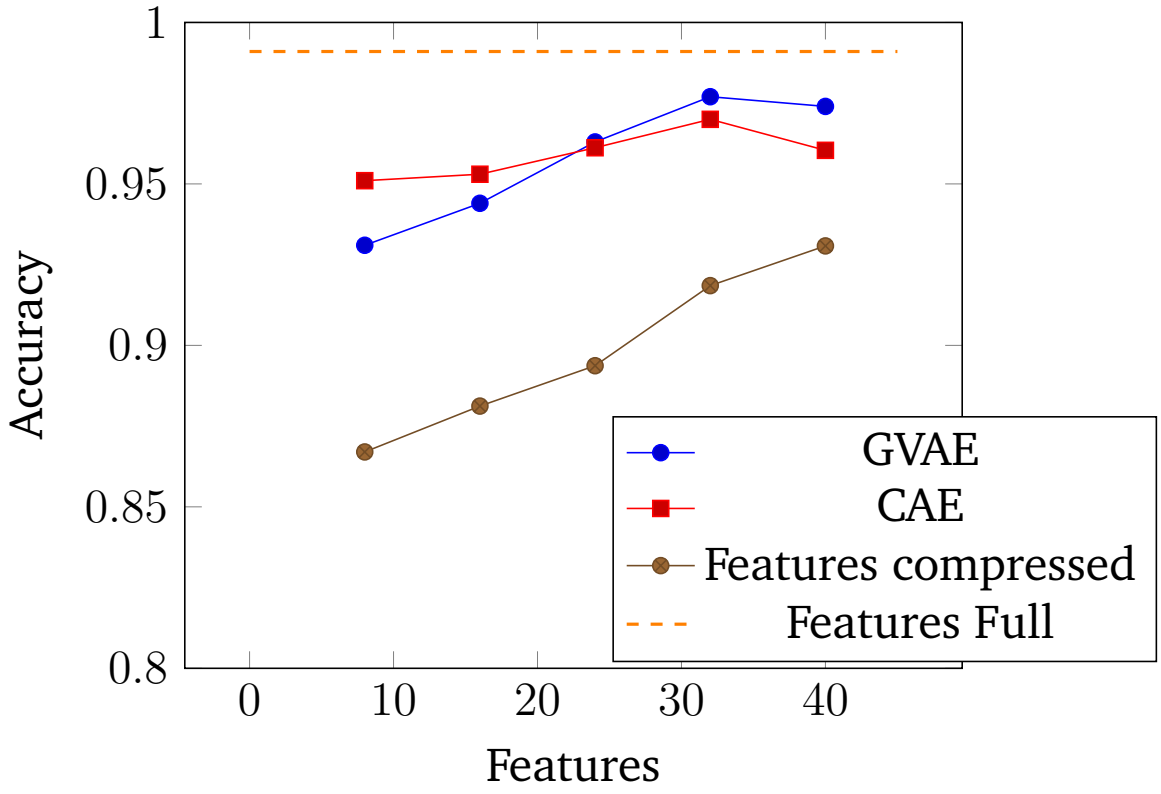


Figure 4.7: SAT/UNSAT classification accuracy.

gressor (RFR) using handcrafted features achieves the highest accuracy across all instance types, with an average accuracy of 99.81%. This indicates that traditional feature-based methods remain highly effective for instance classification. Among the deep learning models, the Graph Neural Network (GNN) outperforms the others with an average accuracy of 94.3%, reflecting its strength in capturing the relational structures of SAT instances. The Binary CNN also shows robust performance with an average accuracy of 88.1%, surpassing the Image CNN's accuracy of 85.2%. This suggests that binary encoding is more suitable for instance classification than image-based encoding. The CAE combined with Random Forest Regressor (CAE + RFR) and the Graph Variational Autoencoder combined with Random Forest Regressor (GVAE + RFR) achieve lower accuracies, 82.6% and 84.5% respectively, indicating that while these models can capture complex latent structures, they may not be as effective as GNNs or CNNs for direct classification tasks.

Table 4.2: Instance classification accuracy

Instance Type	RFR (features)	CNN-IE	CNN-BE	GNN	CAE + RFR	GVAE + RFR
cliquecoloring	99.5%	88.0%	90.2%	94.0%	80.1%	83.2%
count	99.9%	87.5%	91.0%	92.8%	84.0%	85.5%
kclique	99.7%	84.6%	87.8%	93.6%	82.0%	83.8%
kcolor	99.6%	86.2%	88.5%	95.2%	81.5%	84.6%
matching	99.8%	85.0%	87.2%	92.0%	80.3%	82.7%
op	99.9%	83.9%	89.1%	94.5%	83.4%	84.1%
parity	99.7%	84.8%	88.9%	94.1%	82.1%	84.0%
peb	99.6%	86.5%	90.3%	93.9%	81.7%	83.9%
php	99.9%	84.0%	87.4%	94.7%	83.0%	85.1%
ram	99.8%	85.6%	89.3%	94.4%	82.7%	84.9%
stone	99.9%	83.7%	88.0%	95.0%	81.3%	84.8%
subsetcard	99.8%	85.1%	88.2%	93.5%	82.9%	83.5%
tseitin	99.7%	86.2%	89.1%	94.9%	83.6%	85.3%
Average	**99.81%**	**85.2%**	**88.1%**	**94.3%**	**82.6%**	**84.5%**

4.4.4 Model Count Prediction

The results in Table 4.3 provide a comparison of different methods for model counting regression tasks, evaluated using metrics such as Mean Absolute Error (MAE), Root Mean Squared Error (RMSE), R^2 score, and Mean Absolute Percentage Error (MAPE). The Random Forest Regressor (RFR) demonstrates superior performance across all metrics, achieving the lowest MAE (0.125), RMSE (0.173), and MAPE (0.043), along with the highest R^2 score (0.991), indicating its strong capability for accurate model count prediction. The GNN also performs well, with an MAE of 0.176, RMSE of 0.261, and an R^2 score of 0.981, showcasing its effectiveness in leveraging graph representations to predict model counts with high accuracy. In contrast, the deep learning models using image-based encoding (Image CNN) and binary encoding (Binary CNN) show significantly higher errors and lower R^2 scores, with the Image CNN particularly underperforming with an RMSE of 0.981 and MAPE of 0.522. The CAE + RFR combination yields the highest RMSE (1.926) and one of the lowest R^2 scores (0.745), indicating limitations in capturing the necessary features for regression tasks despite its ability to compress data. The GVAE + RFR method, while outperforming the purely image-based and binary CNNs in terms of R^2 score (0.883) and MAPE (0.232), still falls short compared to the GNN and RFR.

Table 4.3: Model Counting Regression accuracy

Method	MAE	RMSE	R2	MAPE
RFR	0.125	0.173	0.991	0.043
CNN-IE	0.876	0.981	0.755	0.522
CNN-BE	0.541	0.764	0.841	0.482
GNN	0.176	0.261	0.981	0.092
CAE + RFR	0.864	1.926	0.745	0.513
GVAE + RFR	0.448	0.864	0.883	0.232

4.4.5 Analysis and Discussion

The observed patterns in the results reflect the strengths and limitations of the different deep learning architectures. Our automatic feature extraction methods, while not achieving the same level of performance as handcrafted features, still demonstrate competitive results, particularly for tasks like SAT/UNSAT classification and model counting. This underscores the advantage of automatic methods, which eliminate the need for in-depth study and manual engineering of potential features, making them more adaptable and scalable to new problem domains.

The superior performance of graph-based models, such as the GNN and GVAE, highlights the importance of graph representations in SAT tasks. These representations effectively encode the relational structures inherent in SAT instances, such as variable dependencies and clause interactions, enabling the models to capture critical information that is essential for accurate predictions. The ability of graphs to encompass these relations gives graph-based methods an edge, particularly for tasks like instance classification and model counting, where understanding structural properties is crucial.

Binary encoding also performs well, as seen in the CNN-BE results, due to the inherent knowledge captured by the convolutional operations applied to the binary representation of SAT instances. This encoding retains much of the instance's logical structure while simplifying the representation, allowing the CNN to learn relevant features efficiently. In contrast, the image-based approach underperforms across all tasks. The reshaping of SAT instances into a 2D grid for image input leads to a loss of key structural and logical information, which significantly impacts the model's ability to extract meaningful patterns.

These findings demonstrate the trade-offs between different encoding methods.

While handcrafted features still hold the advantage for traditional ML models, the ability of automatic methods to generalize across diverse SAT tasks and adapt to new domains without manual intervention positions them as valuable alternatives.

4.5 Conclusions

In this chapter, we explored the application of deep learning models to SAT problems, comparing their performance in SAT/UNSAT prediction, instance classification, and model counting tasks against traditional machine learning methods that rely on handcrafted features. Our study focused on leveraging different deep learning architectures, such as Convolutional Neural Networks (CNNs), Convolutional Autoencoders (CAEs), Graph Neural Networks (GNNs), and Graph Variational Autoencoders (GVAEs), and applying them to distinct SAT instance encodings, including binary, image-based, and graph-based representations.

The results show that deep learning models, particularly those using graph-based techniques like GNNs and GVAEs, have the potential to achieve competitive accuracy in SAT problem prediction tasks. The GNN model demonstrated strong performance across various tasks, leveraging the relational structures of SAT instances effectively. While the traditional Random Forest Regressor (RFR) using handcrafted features generally outperformed deep learning models in instance classification and regression tasks, the deep learning models, especially those leveraging unsupervised feature extraction (CAE + RFR and GVAE + RFR), showed promise in capturing complex patterns that handcrafted features might miss.

Despite these encouraging results, there are several challenges and limitations. The overhead introduced by encoding SAT instances and the computational cost associated with training deep neural networks on large datasets remain significant barriers to the broader applicability of these models. Furthermore, while models like CAEs and GVAEs can effectively learn latent representations, their utility in certain tasks such as instance classification was lower compared to traditional feature-based approaches.

Chapter 5

SAT Generation with Deep Learning Models

Abstract. *This chapter introduces a novel approach for generating realistic and structurally diverse SAT problem instances using a Variational Autoencoder (VAE) combined with a Graph Neural Network (GNN). Our contributions include the development of a VAE-GNN-based SAT instance generator that preserves critical structural properties of SAT problems, a novel compressed representation of the adjacency matrix to enhance model efficiency, and the introduction of new evaluation metrics based on SATzilla's feature extraction tool. We perform an in-depth comparison with existing deep learning SAT generators, evaluating the strengths and weaknesses of each approach. This chapter is based on our previous publication in* ***ESANN: SAT Instances Generation Using Graph Variational Autoencoders*** *[CDOV24] and the code is available.* [1]

5.1 Introduction

Generating realistic and challenging SAT problem instances has become a key focus in advancing SAT solver performance. SAT solvers are often tested on a narrow subset of possible instances, many of which do not reflect the complexity and diversity of real-world problems encountered in industrial or scientific settings. In particular, SAT instances that occur in practice are typically not

[1] https://github.com/mardalla/GVAE2SAT

uniformly distributed across the space of all Boolean formulae, leading to potential poor solver performances when dealing with instances stemming from applications such as software verification, hardware design, and cryptographic analysis. This gap can result in solvers that are overfitted to specific problem types and perform suboptimally when confronted with unfamiliar or more complex instances [Ach09, Spe10].

Randomly generated SAT problems often lack the structural features, like modularity or variable clustering, common in industrial problems, thus providing unrepresentative benchmarks for solver evaluation. Similarly, handcrafted instances, while designed to exploit known weaknesses in solver algorithms, may not generalize well to real-world problems that exhibit more nuanced structures [LENV17]. Consequently, SAT solvers that are trained and tested on such artificial benchmarks may perform poorly in practical applications, underscoring the need for more realistic instance generators.

Developing sophisticated SAT instance generators, particularly those using deep learning models aims to bridge this gap by producing instances that closely resemble real-world SAT problems [GMGC22]. These generators would benefit researchers and practitioners by enabling more comprehensive testing of solvers, ultimately improving their robustness and generalizability. Industries that rely on SAT solvers for essential tasks, such as software verification or electronic design automation, would particularly benefit from the availability of these generated instances, as they more accurately simulate the types of challenges encountered in practice [SLB+18, WR19].

Existing algorithmic approaches for generating SAT problems can be categorized into three broad classes: random, crafted, and industrial SAT generators. Random SAT problems, such as those described by [Ach09], provide a uniform distribution over the possible clauses. However, this leads to instances often lacking the structure found in real-world problems, resulting in unrepresentative benchmarks. Crafted SAT generators like [Spe10] and [EOP20] constrain the generative process by manipulating variables' neighbour relationships to increase solver difficulty, but may produce instances that are still far from those encountered in industrial applications. Lastly, industrial SAT generators, such as [GCL21, ABL22], attempt to simulate real-world instances by focusing on structures such as modularity and scale-free properties but are limited by the need for foreknowledge of problem characteristics.

Given these limitations, there is significant potential in applying deep learn-

ing methods to SAT problem generation. Deep generative models, particularly those using neural networks, can overcome some of the challenges associated with manual problem design by automatically learning structural features from data. This approach enables the generation of problem instances more closely aligned with real-world SAT instances without requiring explicit feature engineering. For example, [BL17] and [SLB+18] demonstrated that graph neural networks (GNNs) could capture critical features of Boolean formulae, leading to better performance in tasks such as SAT classification and satisfiability prediction.

This chapter presents a deep generative model using a variational autoencoder (VAE) combined with a graph neural network (GNN) to generate new SAT instances that reflect the structure of real-world problems. The approach involves representing SAT problem instances as graphs, encoding these graphs using a GNN, and generating new problem instances by sampling from a structured latent space. This method builds on prior works like SATGEN [WR19] and W2SAT [WY23], which use random walks and graph neural networks to generate SAT problems but extends their approaches by using a VAE to ensure a well-structured latent space that allows for meaningful interpolation between generated instances.

Autoencoders have been applied to SAT feature extraction, enabling models to learn lower-dimensional representations of SAT instances [**?**]. However, this work goes beyond simple feature extraction by using a variational autoencoder (VAE), which compresses and reconstructs the input and regularizes the latent space to follow a Gaussian distribution. This procedure allows for the generation of new SAT problem instances through controlled sampling from the latent space.

The contribution of this work is the following:

- A novel SAT instance generator using a VAE-GNN framework that automatically learns and preserves structural features common to real-world SAT problems.
- A novel compressed representation of the adjacency matrix that reduces the computational cost and allows to process much bigger instances.
- An in-depth comparison between different deep learning methods for generating SAT instances, focusing on already established comparison metrics and introducing new metrics developed on the SATzilla [XHHLB08] fea-

ture extraction tool.

Compared to traditional SAT problem generators, our method offers several advantages. First, it eliminates the need for manual feature selection, as the model learns structural characteristics directly from data. Second, it improves upon crafted SAT problems, which may not resemble real-world instances, by generating instances that retain structural features necessary for solver selection and performance, such as modularity and variable clustering, as identified in works like [GCL15] and [AMM22]. Lastly, by providing a generator capable of producing both satisfiable and unsatisfiable instances, this work can help train solvers to handle a more diverse and realistic set of problems, contributing to more robust solver performance in industrial applications.

5.2 Categories of SAT Problems

SAT problems can be classified into several categories based on their structure and origin. These categories have implications for the performance of SAT solvers and the complexity of the instances.

5.2.1 Random and Crafted SAT Problem Generation

Random SAT problems, as defined by [Ach09], are generated by selecting clauses uniformly and independently from all possible combinations of literals. Although random SAT instances can be more challenging than industrial problems on average, their lack of structured dependencies can sometimes result in instances that are easier for solvers to handle, mainly when small backdoors exist with respect to unit propagation [Spe10]. Crafted SAT generators, such as [LENV17], aim to increase solver difficulty by manipulating problem structure, often drawing inspiration from hard combinatorial puzzles or other domains, like graph theory. However, these crafted instances may not always represent the problems encountered in real-world applications.

5.2.2 Industrial SAT Problem Generation

Industrial SAT problems, such as those studied by [GCL15] and [GCL21], often exhibit specific structural features, including high modularity and variable clustering. These structures are essential for simulating real-world problem instances where groups of variables interact in predictable patterns. However,

many current industrial SAT generators focus on reproducing a single structural feature, which limits their ability to fully capture the complexity of real-world SAT problems. To address this limitation, deep learning models offer the potential to automatically learn and reproduce multiple important structural features simultaneously without requiring explicit knowledge of what constitutes a realistic industrial problem.

5.2.3 Deep Learning Approaches to SAT Problem Generation

Recent advances in deep learning, particularly using VAEs and GNNs, have opened new possibilities for SAT problem generation. Unlike traditional methods, deep learning models can learn problem structures directly from data, enabling the generation of instances that retain the complex interactions found in industrial SAT problems. For example, [BL17] and [SLB+18] have demonstrated the effectiveness of GNNs in encoding Boolean formulae, while SATGEN [WR19] and W2SAT [WY23] showed how deep generative models could be used to generate new SAT instances. These models, however, often require careful tuning to avoid issues like mode collapse [RHVL20] and may still fail to generalize across different types of SAT instances.

The method developed in this chapter differs from these earlier works by using a VAE combined with a GNN to encode and decode SAT problem instances directly. This approach avoids some of the complexities associated with GAN-based models while still producing high-quality, challenging SAT problems that reflect the structural features of real-world instances. By focusing on an efficient representation of SAT problems and ensuring a well-structured latent space, this work contributes to the growing field of deep learning approaches to SAT generation, offering new tools for researchers and practitioners alike.

5.3 Method

We outline the methodology employed to generate new SAT problem instances using a graph variational autoencoder (GVAE). The approach involves processing SAT problem instances through the GVAE and evaluating the results. The methodology is divided into four main components: data preparation, model architecture, implementation, and hyperparameter selection.

5.3.1 Data Preparation

Since the aim is to generate realistic propositional formulae, it would be pointless to train on randomly generated Boolean formulae as they do not feature particularly interesting structures and statistical properties that might arise from real world applications. Instead, the dataset consists of problems that resemble real-world applications. The dataset used is the one described in section 3.2.1. Each problem is labelled as either satisfiable or unsatisfiable, with the number of variables ranging from 4 to 250 and the number of clauses from 4 to 2820. All instances are provided in the DIMACS format with metadata included in the header comments.

5.3.1.1 Preprocessing

The GVAE-based model accepts DIMACS files directly, without requiring preprocessing. However, CNFGen sometimes produces instances with empty clauses, which can affect some models. To prevent issues, empty clauses are removed, and headers are updated accordingly. The dataset is then split into training and testing sets. 178 of these instances are chosen as a test set. This subsection was selected such that half of the instances are satisfiable and half unsatisfiable and that a variety of instance sizes be included. Apart from these criteria, the test instances were randomly chosen.

DIMACS files are converted to graph representations suitable for the GVAE by encoding formulae as adjacency matrices of Signed Variable Clause Graphs (SVCGs). Each row represents a clause, and each column represents a variable, with non-zero positions indicating the presence of a variable in a clause. Positive values denote positive literals, and negative values denote negative literals. To reduce size and sparsity, only the populated corners of the full adjacency matrix are used, ensuring bipartiteness and simplifying postprocessing.

The model requires inputs of uniform size. Thus, matrices are padded to match the dimensions of the largest problem: 2820 rows (clauses) and 250 columns (variables). This maximal shape is determined before splitting the dataset to avoid issues during testing.

5.3.1.2 Postprocessing

The GVAE output, a matrix of the same size of the input, is converted back to a DIMACS file. Since the output values are continuous, a threshold is set

to determine literal inclusion. The model allows for user-defined thresholds and adapts them based on the original formula's variable occurrence ratios. Postprocessing steps include:

- Masking to remove padding.
- Ensuring each variable appears in only one clause to avoid trivial unsatisfiability.
- Mapping variables to a continuous range for DIMACS format.
- Writing the DIMACS file with comments linking to the original formula.

5.3.2 Model Architecture

The primary deep learning mechanism we employed was the GVAE, which we explained in Section 4.3.3.3. The architectural design used here is inspired by the approach described by [ZSNL22] in their work on GraphVAE. However, this project does not incorporate their distinctive macro and micro layers, where macro layers capture higher-level community or modular structures and micro layers focus on fine-grained node interactions. Our model specifically targets bipartite Signed Variable-Clause Graphs (SVCGs), where preserving the inherent bipartite structure is crucial. Introducing macro and micro layers, which assume general graph structures, could potentially disrupt this bipartite representation, complicating the model without clear benefit. Thus, we maintain a more streamlined and bipartite-focused architecture for efficiency and clarity.

The framework is composed of an encoder, which projects the input into a latent space, and a decoder, which reconstructs the input from this latent representation. The encoder begins with graph convolutional layers, which are followed by a pooling layer connected to the latent codes via fully connected layers. The decoder then comprises several fully connected layers that regenerate the input based on the latent codes. Further details of the encoder, particularly its graph convolutional network, are discussed below, along with a brief overview of the decoder's structure.

In the encoder, graph convolutional layers, a subset of message passing layers [LC21], are employed. These layers function by propagating information among graph nodes as defined by an input adjacency matrix. The operation of these layers is mathematically described by the formula from [KW16]:

$$\sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}HW) \tag{5.1}$$

where W represents the learnable weights of the layer, and H is the matrix containing the node embeddings that are fed into the layer. The adjacency matrix of the graph is denoted by A. To ensure that each node is also connected to itself, [KW16] add the identity matrix to the original adjacency matrix. $\hat{D}$ refers to the degree matrix of $\hat{A}$, where $\hat{D}_{ii}$ is the number of non-zero elements in row i of $\hat{A}$. The function σ represents the non-linearity applied to the product within the equation, which is essential for enabling learning.

The adjacency matrix $\hat{A}$ is symmetrically normalized by multiplying with $\sqrt{\hat{D}}$ from both sides. This symmetric normalization preserves the graph's structural properties more effectively compared to left multiplication by $\hat{D}^{-1}$, which merely averages neighbor embeddings. Symmetric normalization by $\hat{D}^{-\frac{1}{2}}$ maintains symmetry in the adjacency representation, facilitating a balanced propagation of information and mitigating potential biases arising from nodes with significantly higher degrees.

In a graph convolutional network, the output of one layer, which is the node embedding H, becomes the input for the next layer. The weights W are specific to each layer, while the adjacency matrix remains consistent for all layers processing the same graph. For the first graph convolutional layer, if no specific initial node embeddings are provided, H is typically set to the identity matrix, as done in this project.

A particular challenge is the use of a bipartite adjacency matrix, which prevents the creation of edges between nodes of the same type or between a node and itself. Adding an identity matrix could introduce incorrect connections, so this step is deliberately omitted. Additionally, the adjacency matrix is not necessarily square, which necessitates adjustments such as using only one normalizing matrix (the negated degree matrix) and applying the transpose of the adjacency matrix in alternating layers to maintain dimensional consistency.

For the non-linear activation function, the SELU (Scaled Exponential Linear Unit) [KUMH17] is used, following the tradition of ReLUs [Aga19] but with improved handling of negative values to avoid gradient death during backpropagation.

After the graph convolutional layers, pooling is applied to aggregate node em-

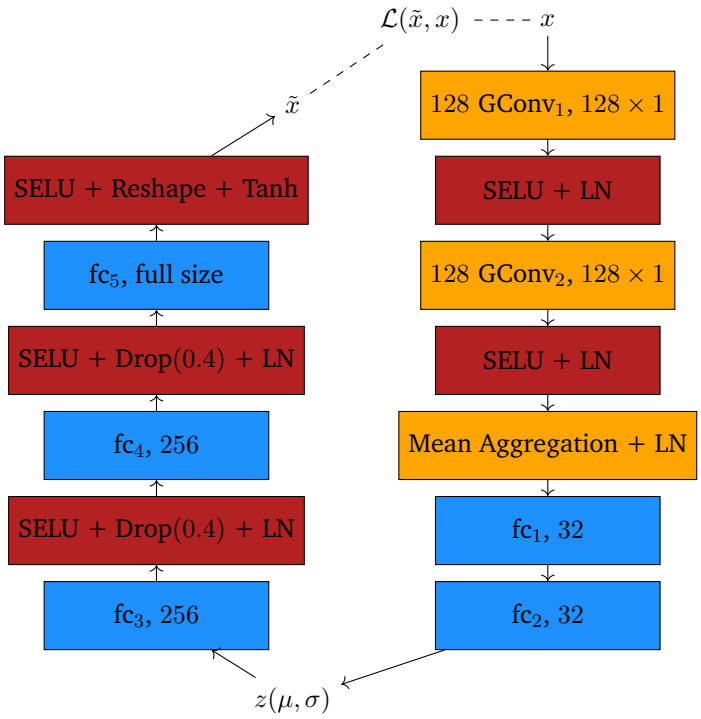


Figure 5.1: Variational Graph Autoencoder consisting of an encoder (right) and decoder (left). The encoder features 2 graph convolutional layers, followed by mean aggregation, and 2 fully connected layers for mean and standard deviation (μ and σ) used to parameterize the latent variable $\mathbf{z}$. The decoder mirrors the encoder structure, with dense layers and appropriate activation, dropout, and normalization layers.

beddings into a single graph embedding. A simple averaging function is used for this purpose, similar to the approach in [ZSNL22]. From this graph embedding, two fully-connected layers generate the latent codes: one for the mean location in the latent space, and another for the noise distribution added during training.

The decoder's role is to regenerate the original input as accurately as possible from the latent code. It is constructed using fully connected layers, with each layer followed by a SELU activation function. To stabilize learning, layer normalization is applied between layers, and dropout is used to prevent overfitting by randomly disabling parts of the network during training. The final layer reshapes the output to match the input dimensions and uses a hyperbolic tangent (tanh) activation function, suitable for outputs in the range of -1 to 1.

5.3.3 Training and Generation

The architecture described was implemented and then executed on NVIDIA Quadro RTX 8000 GPUs. Besides standard Python libraries, the implementation relies on PyTorch [PGM+19] and PyTorch Geometric [FL19] for graph representation learning. PyTorch Geometric's VGAE object was utilized to integrate the encoder and decoder seamlessly and to handle the divergence calculation for latent codes from a normal distribution. Although PyTorch Geometric offers a variety of graph convolutional layers, custom layers were developed specifically to accommodate non-square adjacency matrices.

During the training phase, the dataset was divided into training and testing subsets, with a portion of the training instances further held out as a validation set. Each epoch involved encoding the training problems into the latent space, adding noise to the latent code, and decoding it to regenerate the input. The loss for each iteration was calculated as the sum of the regularization loss (Kullback-Leibler divergence), which is needed to keep the latent space compact and navigable, and reconstruction loss. The reconstruction loss was a combination of losses calculated separately for zero and non-zero elements in the target matrix, weighted by a factor π. The overall loss function is expressed as:

$$z = \lambda_{Recon}(M_0) + \pi \cdot \lambda_{Recon}(M_{\neg 0}) + \beta \cdot \lambda_{Regul}(z) \tag{5.2}$$

where λ_{Recon} represents the reconstruction loss, λ_{Regul} represents the regularization loss, π is the weight applied to the non-zero elements, and β is the weight applied to the regularization loss in the latent space.

Weights were updated after each batch based on the accumulated error, and at the end of each epoch, the model's performance was evaluated on the validation set. Training continued for the predefined number of epochs or until early stopping criteria indicated that further training was unnecessary.

To generate new instances, the trained model remained in evaluation mode, and slight perturbations were applied to the latent codes to generate variations of the input instances. The resulting matrices were postprocessed to obtain CNF formulas in DIMACS format.

Several hyperparameters influenced the model's performance, including the choice of optimizer, loss functions, batch size, and the architecture of layers.

The Adam optimizer was selected due to its adaptive momentum parameter, with a learning rate of 0.03 and a weight decay of 0.1, based on a grid search that tested various combinations of these parameters. The absolute error was chosen as the reconstruction loss to avoid biasing the model toward zero outputs. The β parameter, which controls the balance between the regularization and reconstruction losses, was set to 0.001 after testing different values during early training.

The batch size was determined to be 16 after evaluating various sizes against the validation set. The model's architecture was kept similar to that in [ZSNL22], with two graph convolutional layers in the encoder and three fully connected layers in the decoder. However, the widths of these layers were reduced to explore further compression possibilities.

These hyperparameters were fine-tuned through experimentation, comparing the modularity and clustering of generated graphs with those of the original data instances, to ensure optimal performance of the model.

The GVAE model is trained for 200 epochs, a duration chosen arbitrarily and consistent with the training time of the competitors, which we will present in the next section. Two versions of GVAE are trained: "Allo-GVAE2SAT" on the instance training set without exposure to test instances during training, and "Auto-GVAE2SAT" on the training and test instances combined.

To generate variations, small scalars are added to randomly chosen dimensions of the latent code, producing different formulae. Fig. 5.2 shows the pipeline used for GVAE2SAT.

5.3.4 Competitors

We decided to compare the performance of our model with the following deep-learning SAT instance generators. All evaluated models accept propositional formulae in DIMACS format as input and produce ten more in the same format. Internal representations are ignored due to variations in intermediate encodings among different models. Evaluation focuses solely on the final outputs.

5.3.4.1 EGNN2S and G2SAT

Garzon et al. [GMGC22] introduced both EGNN2S and G2SAT models, exploring techniques to generate realistic SAT instances with models that split nodes

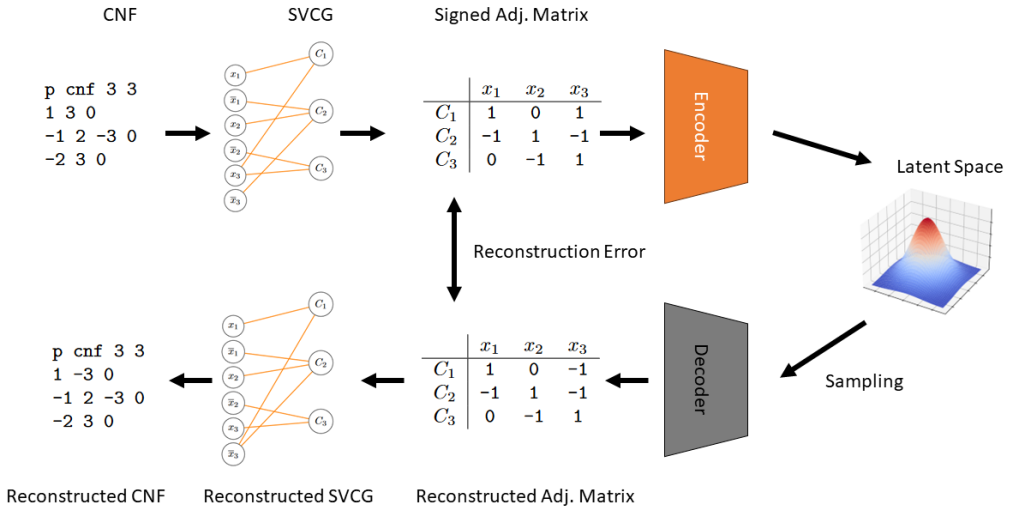


Figure 5.2: Diagram of the pipeline used for GVAE2SAT.

to turn original graphs into several trees. From there, they learn which nodes of the split version should be remerged in order to reconstruct graphs resembling the original. Their study compares node-merging models using different principles in their graph neural layers. G2SAT implements You et al.'s approach [YWB+19], while EGNN2S uses SVCG encodings instead of the original LCG. When [GMGC22] run several of these models against each other, neither of EGNN2S or G2SAT succeeds in outperforming the other on all considered metrics. For this reason, we include both of these best models as competitors in our evaluation. Again, these models generate from the same instances as they train on.

These models require minor modifications for this experiment. Notable changes include:

- Adding functionality to track the origin of regenerated outputs, allowing for accurate mapping between generations and originals. This involved adapting several pipeline functions and resulted in minor inefficiencies, such as learning and saving templates multiple times.
- Modifying the validation step to accommodate instances with pure variables, ensuring all instances are eligible for regeneration despite initial encoding discrepancies.

Both node-merging models are trained stochastically on the test instances be-

fore generating 10 different DIMACS files from each of the 178 instances.

5.3.4.2 W2SAT

W2SAT, developed by Wen et al. [WY23], introduces the *Weighted Literal Incidence Graph (WLIG)* for SAT representation. This model excels in both representation and generation efficiency, capturing more intricate structural and contextual information than previous approaches. The effectiveness of WLIG is further enhanced by a novel hill-climbing optimization method called Optimal Weight Coverage (OWC), which facilitates the efficient translation of WLIGs into SAT problems. W2SAT regenerates Boolean formulae by learning the probability of edge occurrences in graph encodings, training a model to reconstruct graphs by starting with unconnected nodes and intelligently adding edges until a sufficiently well-connected graph is formed.

Some modifications were required to the original implementation of W2SAT:

- Replacing the post-processing step that erroneously set adjacency matrices to zero with a step that prevents literals from co-occurring with their complements, effectively removing tautological clauses.
- Substituting list-based clique enumeration with generator-based enumeration to avoid memory issues, albeit at a potential cost in processing speed.

The model generates new instances based on the same instances it was trained on, creating 10 novel instances per input. However, W2SAT completed only 127 out of the total testing instances due to termination caused by excessive computational cost in some cases.

5.3.5 Metrics

This section describes the metrics used for the assessment of the performances of different generative algorithms.

5.3.5.1 Modularity and Clustering Coefficient

The quality of the generated instances is measured using the modularity and average clustering coefficient of graph encodings in which the generated instances are transformed. These are common metrics in generative graph representation learning.

Modularity. The modularity Q measures the strength of division of a network into communities by comparing the actual edge density inside communities with the density expected in a random graph with the same degree sequence. Formally, for a graph with m edges, adjacency matrix A, degree sequence $\{k_i\}$, and community assignment vector $\{c_i\}$,

$$Q \;=\; \frac{1}{2m}\sum_{i,j}\Big[A_{ij} - \frac{k_i k_j}{2m}\Big]\,\delta\big(c_i, c_j\big), \tag{5.3}$$

where $\delta(c_i, c_j)$ is 1 if nodes i and j are in the same community and 0 otherwise. High values of Q (typically $Q \geq 0.3$ in empirical networks) indicate a pronounced community structure, meaning substantially more intra-community edges than would be expected at random. For a comprehensive treatment, see [CNM04, New06].

Clustering Coefficient. The clustering coefficient captures the tendency of a node's neighbors to be interconnected. For each node i with degree $k_i \geq 2$, let e_i be the number of edges between its k_i neighbors. The *local* clustering coefficient is

$$C_i \;=\; \frac{2e_i}{k_i\big(k_i - 1\big)}. \tag{5.4}$$

The *global* clustering coefficient of the graph can be expressed either as the average of the local coefficients,

$$C_{\text{avg}} \;=\; \frac{1}{n}\sum_{i=1}^{n} C_i, \tag{5.5}$$

or, equivalently, as the ratio of closed triplets (three nodes all mutually connected) to all triplets (three nodes with at least two edges),

$$C_{\text{tri}} \;=\; \frac{\text{number of closed triplets}}{\text{number of all triplets}}. \tag{5.6}$$

Both formulations quantify the graph's propensity for triangle formation—high values signal tightly knit local neighborhoods. Detailed discussions can be found in [WS98, New03].

[GMGC22] evaluates model generations using modularity and clustering coefficient metrics derived from their variable incidence graph (VIG) encodings. [WY23] extend this evaluation to include the modularity of Literal Incidence Graphs (LIG), Variable-Clause Graphs (VCG), and Literal-Clause Graphs (LCG), as well as the average clustering coefficient of LIGs. These metrics are applied to the generated instances. The analysis focuses on the difference between the average metric value of the generations and the original instance, rather than the average value alone, ignoring the variance among a model's ten generations for each instance. For a description of the graphs, refer to Section 4.2.3.

5.3.5.2 Ranking Coefficients

We evaluated the generated instances by using four different solvers and ranking them by the average amount of time it to solve the instances 10 times. To measure the similarity of the rankings, we used Spearman and Kendall rank coefficients.

Spearman's Rank Correlation Coefficient. Spearman's rank correlation coefficient [Zar05], denoted as ρ, is a non-parametric measure of rank correlation. It assesses how well a relationship between two variables can be described using a monotonic function, meaning it evaluates whether the rank order of one variable increases or decreases with the rank order of another. This method is particularly advantageous when the data are ordinal or when the assumptions of linear correlation (as in Pearson's correlation) are not met. Unlike Pearson's correlation, Spearman's ρ does not require the data to be normally distributed or the relationship between variables to be linear. This makes ρ a versatile tool in analyzing non-linear relationships where only the ranks of data are of interest. The formula for calculating Spearman's ρ is given by:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)}$$

where d_i represents the difference between the ranks of each pair of corresponding values and n is the number of observations.

Kendall's Tau. Kendall's Tau [Puk11], often symbolized as τ, is a statistic used to measure the ordinal association between two quantities. It provides an assessment of the strength and direction of the relationship between two ranked variables. This measure is particularly useful when dealing with ordinal data

or when the assumption of a continuous distribution for Pearson's correlation cannot be justified. Kendall's Tau is considered more robust and less affected by ties in data than Spearman's ρ. The computation of τ focuses on the concordance and discordance between pairs of data points, essentially comparing the ranks within each pair to assess their correlation. The formula for Kendall's τ is:

$$\tau = \frac{2}{n(n-1)} \sum_{i<j} \text{sgn}(x_i - x_j) \cdot \text{sgn}(y_i - y_j)$$

where n is the number of pairs, and x_i, x_j, and y_i, y_j are the ranks of observations in each variable, with sgn denoting the sign function.

5.3.5.3 SATzilla Feature Distance

A final evaluation involves comparing the generated instances to the original ones using SATzilla features [XHHLB08]. SATfeatPy [PDVO22] extracts 33 features, which are reduced to two dimensions using t-SNE, a dimensionality reduction technique particularly well-suited for visualizing high-dimensional data. t-SNE effectively condenses the 33-dimensional feature space into a more interpretable two-dimensional space, allowing for a clearer comparison between the original and generated instances.

5.4 Evaluation

This section evaluates the GVAE by comparing its performance to other models designed to generate new SAT problem instances from original ones using graph neural networks. All models tried to generate 10 instances from each of these 178 test instances. For our experimentation, W2SAT was set to time out and abandon generating from an instance to avoid exhausting the available RAM. For this reason, W2SAT did not provide any results for 51 instances. Figure 5.3 shows the distribution of instance type and satisfiability of the two partitions of the test dataset. We can see that W2SAT fails in clique and dominating instances.

Figures 5.10 show the results. Each vertical line corresponds to a problem instance, with the position of a model on the line indicating the difference between the average value of its generations and the original instance for the specified metric.

These visualizations suggest that W2SAT consistently achieves the closest met-

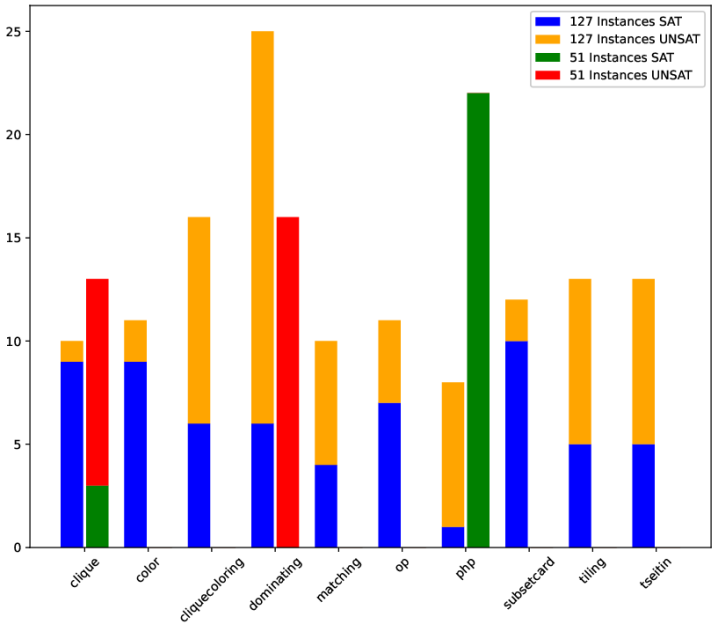


Figure 5.3: Distribution of the test instances

rics to the original across all measures. The node-merging models (EGNN2S and G2SAT) appear to compete for second place, while the GVAE models generally perform the worst, except possibly in the modularity of VIGs, where the GVAE trained on the larger dataset may rival EGNN2S. Interestingly, the GVAE models seem to replicate the average clustering coefficient of VIGs better than that of LIGs, potentially indicating difficulties in learning polarity or simply reflecting that the original VIGs' average clustering coefficients are generally closer to 0.5, thereby limiting the maximum possible error.

Table 5.1 present the results in a more concise form. The results are divided into the two subset of the testing set depending on whether W2SAT was able to produce results. The "Avg. Error" represents the mean error across all instances, averaged over the ten generations for each model. These tables reinforce the advantage of W2SAT, although the GVAE models occasionally achieve better averages on certain instances compared to the other models.

Table 5.1: Modularity and clustering coefficient for the graphs of the generated instances

Metric & Model	127 Instances		51 Instances	
	Avg. Error	# Best	Avg. Error	# Best
Clustering Coefficient				
LIG				
EGNN2S	0.20	35	0.35	44
G2SAT	0.29	23	0.45	7
W2SAT	0.13	66	-	-
Allo-GVAE2SAT	0.44	2	0.71	0
Auto-GVAE2SAT	0.44	11	0.71	0
VIG				
EGNN2S	0.16	23	0.13	38
G2SAT	0.22	15	0.21	13
W2SAT	0.06	73	-	-
Allo-GVAE2SAT	0.29	11	0.51	0
Auto-GVAE2SAT	0.29	5	0.51	0
Modularity				
LCG				
EGNN2S	0.13	23	0.17	21
G2SAT	0.16	9	0.16	23
W2SAT	0.04	73	-	-
Allo-GVAE2SAT	0.12	12	0.15	4
Auto-GVAE2SAT	0.12	10	0.15	3
LIG				
EGNN2S	0.24	7	0.30	13
G2SAT	0.25	7	0.22	22
W2SAT	0.03	106	-	-
Allo-GVAE2SAT	0.22	4	0.34	9
Auto-GVAE2SAT	0.21	3	0.34	7
VCG				
EGNN2S	0.14	21	0.20	20
G2SAT	0.18	18	0.19	22
W2SAT	0.04	71	-	-
Allo-GVAE2SAT	0.17	9	0.16	5
Auto-GVAE2SAT	0.17	8	0.16	4
VIG				
EGNN2S	0.36	4	0.36	17
G2SAT	0.26	13	0.18	6
W2SAT	0.05	92	-	-
Allo-GVAE2SAT	0.24	11	0.20	15
Auto-GVAE2SAT	0.24	7	0.20	13

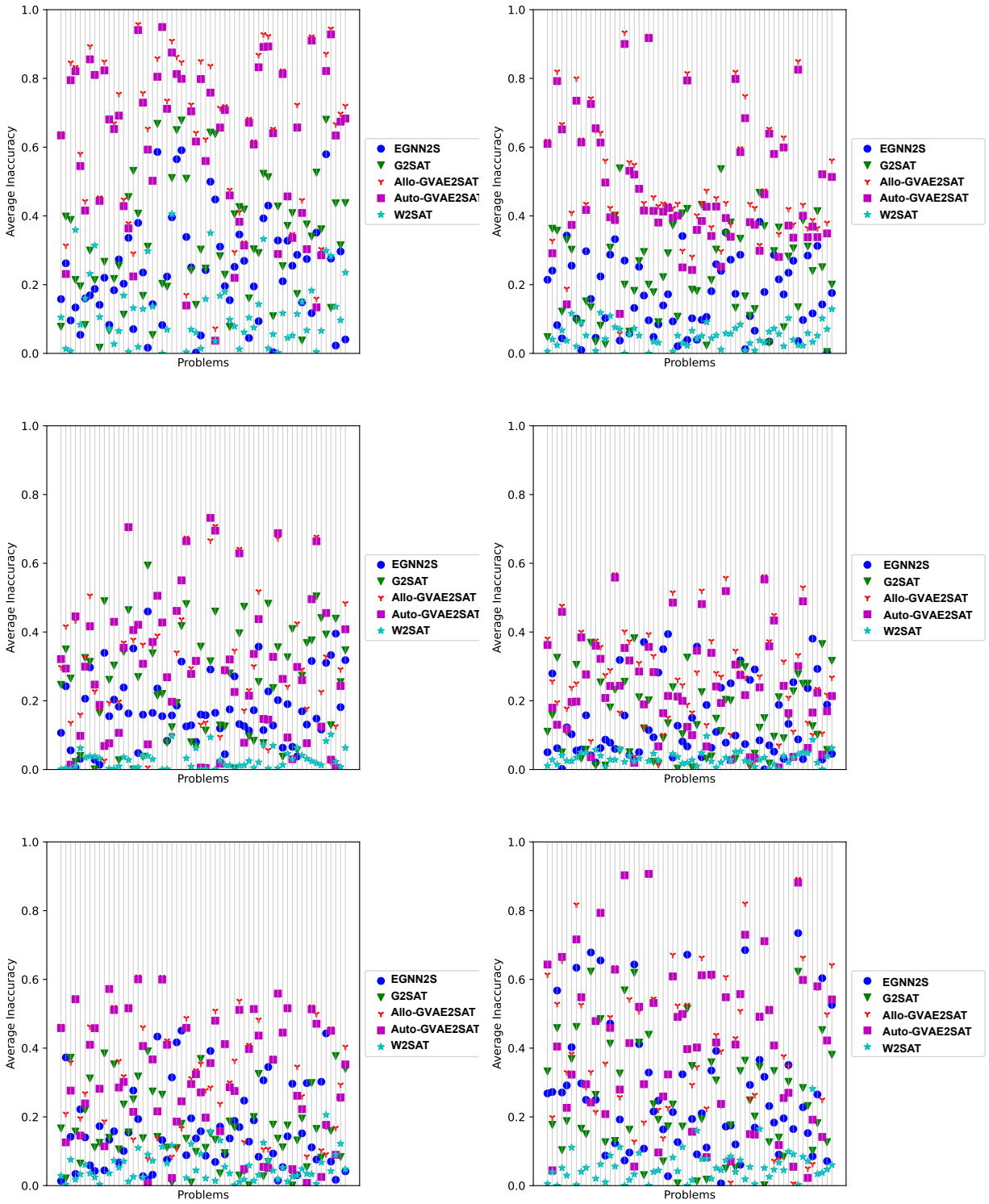


Figure 5.10: Modularity and clustering coefficient for the graphs of the generated instances.

5.4.1 Solver Performance

This subsection evaluates the similarity between the generated and original instances based on their behavior with SAT solvers. Four solvers are used: CaDiCaL 1.5.3, Glucose 4.2.1, Lingeling, and MapleCOMSPS_LRB.

Satisfiability. The capacity to generate satisfiable SAT instances is a pivotal aspect of SAT instance generators, yet it remains a significant challenge as highlighted in recent studies [GMGC22, WY23]. Table 5.2 reveals the performance of various models in generating satisfiable instances. GVAE2SAT exhibits the highest rate of generating satisfiable instances from SAT seeds (40%), indicating its effectiveness in preserving satisfiability. Conversely, it also flips a substantial number of previously UNSAT instances (24%), demonstrating a notable discrepancy in performance between SAT and UNSAT seeds. In contrast, W2SAT shows the lowest rate of flipping UNSAT instances, suggesting better consistency between SAT and UNSAT transformations. The peculiarities in the 51 instances dataset result in only a few satisfiable outputs from GVAE2SAT, underscoring the model's sensitivity to specific formula characteristics.

Table 5.2: Percentage of satisfiability of generations

	127 instances		**51 instances**	
Model	**SAT**	**UNSAT**	**SAT**	**UNSAT**
EGNN2S	15.3%	19.0 %	0.0 %	0.0 %
G2SAT	14.5%	13.9%	0.0 %	0.0 %
W2SAT	15.3 %	12.8 %	-	-
Allo-GVAE2SAT	32.3%	21.8%	0.0%	0.2%
Auto-GVAE2SAT	40.3%	23.7%	0.0%	0.4%

The recall of satisfiability is generally poor for all models, with the GVAE models performing the best.

Solver Performance. Another sensible, domain-specific evaluation of regenerations is measuring how closely their relationship with SAT solvers resembles that of their original instances, in particular, if the instances have a similar difficulty. This subsection summarizes the findings of deploying on the original and generated problems four solvers provided in PySat[IMM18]: CaDiCaL 1.5.3, Glucose 4.2.1, Lingeling, and MapleCOMSPS_LRB. We consider the average computational time a solver requires to tackle an instance over 10 repetitions

as a measure of its hardness. We then investigate if the synthetic instances preserve the rank among the solvers; this is a widely used metric in SAT instances generation.

Table 5.3 shows the average computational times and the Spearman and Kendall coefficients. The correlation coefficients are computed for the order of the solvers on the *instances*, and for the rank of time required for each instance for the specific *solver*. We can see that the generated instances are between 3 and 5 orders of magnitude faster to solve compared to the original CNFs; this result is in line with the findings of [GMGC22]. G2SAT generates the more challenging instances, while our approach and W2SAT struggle to create complex ones. It is interesting that all generators quite well preserve the rank among solvers, but this is generally due to inherited differences in solvers' speed, e.g. Maplesap is faster than Lingeling in most instances. Analysing the performances of the specific solver, GVAE2SAT has the higher score, but all the generators struggle to preserve the ranking. The correlation values are close to a random shuffle of the order. Overall, no deep learning approach generates instances that are of similar complexity for SAT solvers.

Table 5.3: Average Spearman and Kenadall ranking metrics of the solving times for the generated instances compared to the solving times for the new instances. The computational time is provided by PySat; the solvers' initialization clause load is not considered.

		Original	**Allo-GVAE2SAT**	**Auto-GVAE2SAT**	**G2SAT**	**EGNN2S**	**W2SAT**
Computational Time		3.406	$1.94{\cdot}10^{-5}$	0.001	0.002	$2.11{\cdot}10^{-4}$	$2.57{\cdot}10^{-5}$
Instance	ρ	1	0.551	0.560	0.584	0.433	0.587
	τ	1	0.412	0.457	0.475	0.339	0.504
Solver	ρ	1	0.1261	0.1545	0.034	0.084	0.047
	τ	1	0.994	0.106	0.025	0.059	0.033

Regarding the computational times, our analysis confirms that the algorithms struggle to synthesize hard instances. G2SAT generations are the ones that the solvers struggle more with, creating instances of similar difficulty for the 51 set; however, there are three orders of magnitude differences with the original instances for the 127 set. The synthetic formulae generated by the GVAE2SAT and W2SAT are considerably easier for a solver. This is because their generations include unit clauses that contradict themselves or have a clear solution. Moreover, GVAE2SAT generates more satisfiable formulas, and for small instances, it is generally easier to find a solution instead of proving the non-existence.

5.4.2 Generation Computational Times

The scalability of the different approaches is a crucial aspect to evaluate their applicability to relevant scenarios. The dataset used herein comprises small instances compared to real-world problems or formulae used in SAT competitions [BFH+20]. Table 5.4 shows the computational time required by the different phases of the generative models. Considering the overall time required, Auto-GVAE2SAT is between 69 and 288 times faster than the competitors. However, GVAE2SAT differs from the other models since, after the training phase, it can obtain new instances from the same distribution and return synthetic CNFs without retraining. This makes it able to generate the instances (10 variations for each test problem) in a bit more than a minute without the need for re-training. The other approaches need to be trained on the specific test instance before creating new formulae, making them considerably slower in the extension of existing datasets.

The high computational effort required by these approaches has been a limiting aspect of the deployment of these techniques. [GMGC22] experiments required more than 240 days of computations, while [WY23] did not present a computational time comparison with other approaches. The algorithm presented herein can generate instances of a fraction of the time compared to the existing approaches.

Table 5.4: Running Times (s)

	Pre.	**Training**	**Generation**	**Post.**	**Total**
EGNN2S	0.46	14503.75	11407.81	130.81	26042.83
G2SAT	0.38	3432.73	7388.74	126.93	10948.78
W2SAT		1509.34	115595.02		117104.36
Allo-GVAE2SAT	0.003	12935.81	18.40		12954.21
Auto-GVAE2SAT	0.0007	190.72	13.67		204.39

5.4.3 SATzilla Feature Distance

Table 5.5 presents the mean L1 and L2 distances between the original and generated instances in this reduced space, providing a metric for evaluating their similarity.

W2SAT performs best in reproducing SATzilla features, followed by G2SAT, with the GVAE models trailing behind.

Table 5.5: Error of SATzilla Features Reduced with t-SNE

	127 instances		51 instances	
Model	**L1**	**L2**	**L1**	**L2**
EGNN2S	21.97	18.50	21.36	17.96
G2SAT	1.63	1.25	1.60	1.22
W2SAT	0.64	0.49	-	-
Allo-GVAE2SAT	24.08	19.90	22.77	18.78
Auto-GVAE2SAT	23.90	19.92	22.54	18.76

Overall, these evaluations highlight the superior performance of W2SAT in generating realistic SAT instances, with G2SAT as the second-best model. While the GVAE models lag in most metrics, they still contribute valuable insights into the generation of SAT problems using graph-based approaches. Future work should focus on refining these models to improve their performance and better understand their limitations.

The comprehensive evaluation conducted in this section sheds light on the comparative performance of various models in generating new SAT problem instances. The key findings are:

- **Structural Metrics:** W2SAT consistently outperforms other models in terms of modularity and average clustering coefficient, closely replicating the structural properties of original instances. GVAE models, while trailing behind, demonstrate potential areas for improvement, particularly in reproducing VIG clustering coefficients.
- **Solver Performance:** The GVAE models generate instances that are generally easier to solve compared to those produced by other models, as indicated by average solving times. Despite their ease, GVAE-generated instances maintain relative solver difficulty rankings, showcasing some fidelity to original instance complexity.
- **Satisfiability Recall:** GVAE2SAT is highly effective in preserving satisfiability, generating 40% satisfiable instances from SAT seeds, but it also shows a significant discrepancy by flipping 24% of previously UNSAT instances. This highlights a significant challenge in generating satisfiable instances that mirror the original problem's properties.
- **SATzilla Features:** When comparing the generated instances to the originals using SATzilla features, W2SAT and G2SAT models excel, indicating

their effectiveness in capturing a broader range of structural characteristics. GVAE models show higher errors in these features, suggesting areas for further development.

5.5 Conclusions

This chapter introduced a novel approach for generating SAT problem instances using a Variational Autoencoder combined with a Graph Neural Network. We developed the GVAE2SAT model to address limitations in existing SAT generators by capturing and preserving structural features of SAT instances. Unlike traditional random and crafted generators, the GVAE framework automatically learns structural properties directly from data. We compared the GVAE2SAT model against state-of-the-art generative approaches, including W2SAT, EGNN2S, and G2SAT, across multiple evaluation metrics. Structural metrics such as modularity and clustering coefficients demonstrated that W2SAT consistently outperformed other models, while GVAE2SAT exhibited competitive performance in selected graph encodings, particularly for VIG clustering coefficients. Regarding satisfiability recall, GVAE2SAT achieved the highest rate of generating satisfiable instances but showed some inconsistency when handling unsatisfiable inputs. An evaluation based on solver performance revealed that all generative approaches produced significantly easier instances to solve than the originals. However, GVAE2SAT and its competitors successfully preserved solver rankings, ensuring relative consistency in solver behaviour across generated and original instances. Furthermore, comparisons using SATzilla features confirmed the superior fidelity of W2SAT and G2SAT in generating instances that reflect original problem characteristics, with GVAE2SAT trailing behind. Finally, we observed that GVAE2SAT significantly outperformed other methods in terms of speed and scalability of instances generation. Unlike its competitors, GVAE2SAT does not require retraining for new instances from the same distribution, as well as minimal postprocessing of the instances, making it highly efficient for extending datasets. In summary, the GVAE2SAT model represents an important step toward scalable and efficient SAT instance generation while providing new insights into the structural and solver-related properties of generated problems. This chapter explores in depth the challenges and opportunities in applying deep learning algorithms to SAT generation, contributing to the ongoing development of realistic and diverse SAT benchmarks for solver evaluation.

Chapter 6

Conclusions and Future Works

6.1 Conclusions

The research presented in this thesis addresses key challenges in the field of Boolean Satisfiability (SAT) and Model Counting (#SAT) problem-solving by leveraging machine learning techniques to enhance accuracy at predicting whether a SAT instance is satisfiable, its category and its model count. SAT has been a fundamental problem in computer science and artificial intelligence, with applications ranging from hardware verification to cryptography. The thesis focuses on developing machine learning-based methods to provide new approaches for tackling SAT problems, particularly in feature extraction, SAT/UNSAT prediction, instance classification, model counting, and instance generation.

In Chapter 3, we developed a machine learning framework that utilized diverse features extracted for predicting SAT/UNSAT, instance classification and model counting. We introduced a tool that can extract a wide range of features that capture the structural and statistical properties of SAT instances. An essential contribution of this chapter was the successful application of machine learning models, specifically Random Forest and XGBoost, to predict SAT/UNSAT, instance type, and estimate model counts. Additionally, we incorporated explainability techniques, such as SHAP (SHapley Additive exPlanations), to interpret the importance of features and provide insights into the decision-making process of the models. Experimental results demonstrated that these models achieved high accuracy in SAT/UNSAT prediction, instance classification and provided reliable model count estimates, significantly reducing computational time compared to traditional methods. This approach proved effective across

multiple datasets, highlighting both the predictive power and the explainability of feature-based machine learning models in enhancing SAT analysis techniques.

In Chapter 4, we explored the application of deep learning models, such as Convolutional Neural Networks (CNNs), Graph Neural Networks (GNNs), Convolutional Autoencoders (CAE) and Graph Variational Autoencoders (GVAEs), for SAT/UNSAT classification, instance classification, and model counting. These models were applied to different encodings of SAT instances, including binary, image-based, and graph-based representations, allowing for automatic feature extraction and leveraging the relational structures in SAT problems. The main contribution was demonstrating the effectiveness of GNNs and GVAEs in capturing the structural complexity of SAT problems, with GNNs achieving high accuracy in both SAT/UNSAT classification and model counting tasks. Experimental results showed that while traditional machine learning models still performed strongly, deep learning models provided a competitive advantage by eliminating the need for manual feature engineering and generalizing effectively across diverse datasets.

In Chapter 5, we developed a deep generative model for producing SAT problem instances that resemble real-world challenges. The model leverages Graph Variational Autoencoders to generate new SAT instances by encoding Boolean formulae as variable-clause graphs. This combination allows the model to capture both the local and global structures of SAT problems, enabling the generation of instances that maintain the structural properties crucial for testing modern SAT solvers. Unlike other deep learning competitors, our approach does not require any post-processing steps and is generally faster, making it more efficient for large-scale SAT instance generation. Experimental results demonstrated that the generated instances preserved key features such as modularity and clustering coefficients, preserve satisfiability ensuring their similarity to the original SAT problems. This approach provides a basis for a flexible and efficient method for generating diverse SAT problems, addressing the limitations of traditional problem-generation methods that rely on handcrafted rules.

The work presented in this thesis contributes to the growing field of machine learning applications for SAT problem-solving by introducing new methods for feature extraction, predictive modelling, and instance generation. Each of these contributions offers potential improvements for SAT solvers, enhancing both their accuracy and efficiency.

In conclusion, this thesis presents a series of contributions to the SAT and #SAT solving landscape, offering new methods for integrating machine learning with traditional solving approaches. The methods proposed here have the potential not only to advance theoretical research in this area but also provide practical benefits for industries relying on SAT solvers for complex problem-solving tasks.

6.2 Future Work

While this thesis made substantial progress in applying machine learning to SAT problems, several areas remain open for further exploration, which could significantly enhance the applicability and effectiveness of these approaches.

6.2.1 Improved Variable Selection and Branching Heuristics

The use of machine learning classifiers for branching heuristics, as explored in [BDV+23], has shown promising results by leveraging SAT/UNSAT classifiers to guide variable selection. However, the absence of backtracking limits the current implementation's efficiency. Future research should focus on integrating backtracking mechanisms and exploring advanced machine learning techniques, such as reinforcement learning, to dynamically adapt branching strategies [YP19, WR18]. These methods could further optimize solver performance by tailoring variable selection to the specific characteristics of SAT instances in real-time.

6.2.2 Deep Learning-Driven SAT Solvers

While deep learning models like GNNs have demonstrated exceptional performance in tasks such as satisfiability prediction [SLB+18, CZL22], their integration into real-world SAT solvers is an area ripe for exploration. Developing end-to-end SAT solvers with embedded deep learning components could enable solvers to make dynamic, data-driven decisions during the search process. For instance, integrating learning-based heuristics for variable selection and clause deletion into CDCL solvers could significantly reduce solving time by narrowing the search space [ZSZ+20, KB23].

6.2.3 Advancements in SAT Problem Generation

The Graph Variational Autoencoder (GVAE) framework introduced in this thesis has proven effective for generating structurally diverse SAT instances. However, future work should aim to refine the generative process to better emulate industrial problem structures, such as those in hardware verification and cryptographic analysis [GCL21, AMM22]. Integrating SAT-solving performance metrics into the generation process could yield instances that not only resemble real-world problems but also serve as more rigorous benchmarks for solver evaluation [WR19, WY23].

6.2.4 Scalability and Generalization of Machine Learning Models

Scaling machine learning models to handle larger and more complex SAT instances is a critical challenge. Distributed training frameworks and efficient feature extraction methods could help manage the computational demands associated with larger datasets [Ach09, Spe10]. Moreover, ensuring generalization across diverse SAT instance types requires expanding datasets to include industrial and highly structured problems. Such diversification could improve model robustness and applicability, making them better suited for real-world scenarios [SGBP20].

6.2.5 Explainability and Interpretability in SAT Solvers

As machine learning models become more integrated into SAT solvers, the need for explainability and interpretability grows. Future research should explore methods to make these models more transparent, particularly in explaining how certain features influence predictions. Explainable AI (XAI) [R$^+$16] techniques could be applied to both feature-based and deep learning models to provide insights into their decision-making processes, which could be useful for further optimizing SAT solvers.

6.2.6 Refinement of Latent Representations

Deeper exploration of the latent spaces generated by models like CAEs and GVAEs could reveal how their dimensions correspond to traditional SAT features. Such analyses could uncover blind spots in existing feature sets and

inspire the development of new, more informative features [SLB+18, BL17]. Tailoring reconstruction losses specifically for SAT instances to preserve structural characteristics could further improve generative model quality.

6.2.7 Hybrid Feature Sets and Novel Encodings

Combining handcrafted statistical features with deep learning-extracted features could lead to comprehensive feature sets that capture multiple facets of SAT. Additionally, exploring alternative encoding methods—such as hybrid encodings that blend binary, graph-based, and image-based representations—could maximize information retention and minimize computational complexity, thereby enhancing performance across various SAT tasks [WYC+19].

6.2.8 Integration of Generative Models with Solvers

Integrating generative models with SAT solvers could enable an iterative feedback loop, where solvers evaluate generated instances and inform model refinement. This end-to-end framework could lead to the creation of increasingly challenging and realistic benchmarks, improving solver robustness and adaptability [WR19, WY23].

6.2.9 Broader Applications of Generative Models

The methodologies developed for SAT instance generation could extend beyond SAT problems to other domains, such as combinatorial optimization, constraint satisfaction, and machine learning benchmarks [CBZ+19]. Exploring these cross-domain applications could provide valuable new tools, for instance, generation and problem-solving in various fields.

6.2.10 Algorithm Portfolios for Model Counting

Future research could focus on developing algorithm portfolios specifically tailored for model counting tasks. Such portfolios would dynamically select the most appropriate model counting algorithm based on the structural and statistical characteristics of the SAT instance. This approach could leverage the strengths of various algorithms, such as exact counters for simpler instances and approximate methods for more complex ones. Recent work [Dun24] has emphasized the variability in performance across different model counters, high-

lighting the potential for significant improvements through portfolio-based strategies. By incorporating predictive models to assess instance complexity and runtime behaviour, these portfolios could optimize performance and broaden the applicability of model counting techniques in both theoretical and practical domains.

References

[A+20] Alejandro Barredo Arrieta et al. Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges toward Responsible AI. *Information Fusion*, 58:82–115, 2020.

[ABGCL17] Carlos Ansótegui, Maria Luisa Bonet, Jesús Giráldez-Cru, and Jordi Levy. Structure features for SAT instances classification. *Journal of Applied Logic*, 23:27–39, 2017.

[ABL22] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Scale-Free Random SAT Instances. *Algorithms*, 15(6):219, 2022.

[Ach09] Dimitris Achlioptas. Random Satisfiability. *Handbook of Satisfiability*, 185:245–270, 2009.

[Aga19] Abien Fred Agarap. Deep Learning using Rectified Linear Units (ReLU), 2019.

[AM14] Enrique Matos Alfonso and Norbert Manthey. New CNF Features and Formula Classification. In *POS@ SAT*, pages 57–71, 2014.

[AM18] Naveed Akhtar and Ajmal Mian. Threat of Adversarial Attacks on Deep Learning in Computer Vision: A Survey. *IEEE Access*, 6:14410–14430, 2018.

[AMM22] Tasniem Nasser Alyahya, Mohamed El Bachir Menai, and Hassan Mathkour. On the Structure of the Boolean Satisfiability Problem: A Survey. *ACM Computing Surveys (CSUR)*, 55(3):1–34, 2022.

[AS95] Khaled S Al-Sultan. A Tabu search approach to the clustering problem. *Pattern recognition*, 28(9):1443–1451, 1995.

[AS09] Gilles Audemard and Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers. In *IJCAI*, 2009.

[AS12] Gilles Audemard and Laurent Simon. Refining restarts strategies

for SAT and UNSAT. *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 107–121, 2012.

[ASA+23] Farah Liyana Azizan, Saratha Sathasivam, Majid Khan Majahar Ali, Nurshazneem Roslan, and Caicai Feng. Hybridised Network of Fuzzy Logic and a Genetic Algorithm in Solving 3-Satisfiability Hopfield Neural Networks. *Axioms*, 12(3):250, March 2023.

[B+20] Tom B. Brown et al. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[BBK89] Franc Brglez, David Bryan, and Krzysztof Koiminski. Combinational profiles of sequential benchmark circuits. *IEEE International Symposium on Circuits and Systems,*, pages 1929–1934 vol.3, 1989.

[BCDZ17] Alessio Benavoli, Giorgio Corani, Janez Demšar, and Marco Zaffalon. Time for a Change: a Tutorial for Comparing Multiple Classifiers Through Bayesian Analysis. *Journal of Machine Learning Research*, 18(77):1–36, 2017.

[BDKH20] André Biedenkapp, Nguyen Dang, Martin S. Krejca, and Frank Hutter. Dynamic Algorithm Configuration: Foundation of a New Meta-Algorithmic Framework. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI)*, pages 427–434, 2020.

[BDP03] Fahiem Bacchus, Shannon Dalmao, and Toniann Pitassi. Algorithms and Complexity Results for #SAT and Bayesian Inference. In *Proceedings of FOCS*, pages 340–351, 2003.

[BDV+23] Ruth Helen Bergin, Marco Dalla, Andrea Visentin, Barry O'Sullivan, and Gregory Provan. Using machine learning classifiers in SAT branching. In *Proceedings of the International Symposium on Combinatorial Search*, volume 16, pages 169–170, 2023.

[BFH+20] Tomáš Balyo, Nils Froleyks, Marijn JH Heule, Markus Iser, Matti Järvisalo, and Martin Suda. Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions. 2020.

[BG18] Joy Buolamwini and Timnit Gebru. Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification.

In *Proceedings of the 1st Conference on Fairness, Accountability and Transparency*, pages 77–91, 2018.

[BHN19] Solon Barocas, Moritz Hardt, and Arvind Narayanan. Fairness and Machine Learning. *Fairness and Machine Learning*, 2019.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*, volume 185. IOS press, 2009.

[Bie08] Armin Biere. Adaptive restart strategies for conflict driven sat solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing – SAT 2008*, pages 28–33, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[BKK+16] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, et al. Aslib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, 2016.

[BKS04] Paul Beame, Henry Kautz, and Ashish Sabharwal. Understanding the power of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.

[BL17] Benedikt Bünz and Matthew Lamm. Graph Neural Networks and Boolean Satisfiability. *arXiv preprint arXiv:1702.03592*, 2017.

[Bol76] Béla Bollobás. Complete subgraphs are elusive. *Journal of Combinatorial Theory, Series B*, 21(1):1–7, 1976.

[Bre01] Leo Breiman. Random Forests. *Machine learning*, 45:5–32, 2001.

[BS22] Filip Beskyd and Pavel Surynek. Parameter Setting in SAT Solver using Machine Learning Techniques. In Ana Paula Rocha, Luc Steels, and H. Jaap van den Herik, editors, *Proceedings of ICAART 2022*, pages 586–597, 2022.

[C+15] François Chollet et al. Keras. `https://keras.io`, 2015.

[CBZ+19] Di Chen, Yiwei Bai, Wenting Zhao, Sebastian Ament, John Gre-

goire, and Carla Gomes. Deep Reasoning Networks: Thinking Fast and Slow. *arXiv preprint arXiv:1906.00855*, 2019.

[CDOV24] Daniel Crowley, Marco Dalla, Barry O'Sullivan, and Andrea Visentin. SAT Instances Generation Using Graph Variational Autoencoders. In *ESANN 2024 Proceedings*, ESANN 2024, page 369–374. Ciaco - i6doc.com, 2024.

[CG16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016.

[CKJ08] Jianer Chen, Iyad A Kanj, and Weijia Jia. Parameterized approximation hardness of the clique and dominating set problems. *SIAM Journal on Computing*, 37(5):1673–1695, 2008.

[CMV16] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 3569–3576, 2016.

[CN10] Stephen Cook and Phuong Nguyen. *Logical Foundations of Proof Complexity*. Cambridge University Press, January 2010.

[CNM04] Aaron Clauset, M E J Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E*, 70(6):066111, 2004.

[Com21] European Commission. Proposal for a Regulation of the European Parliament and of the Council Laying Down Harmonised Rules on Artificial Intelligence (Artificial Intelligence Act), 2021. Available at `https://artificialintelligenceact.eu/the-act/`.

[Coo71] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[CS12] Shaowei Cai and Kaile Su. Configuration checking with aspiration in local search for SAT. *AAAI*, 2012.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Machine Learning*, 20(3):273–297, 1995.

[CW17] Nicholas Carlini and David Wagner. Adversarial Examples Are Not Easily Detected: Bypassing Ten Detection Methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 3–14, 2017.

[CZL22] Wenjing Chang, Hengkai Zhang, and Junwei Luo. Predicting propositional satisfiability based on graph attention networks. *International Journal of Computational Intelligence Systems*, 15(1), September 2022.

[Dar01] Adnan Darwiche. Decomposable negation normal form. *Journal of the ACM (JACM)*, 48(4):608–647, 2001.

[DBB17] Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for sat. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017*, pages 83–100, Cham, 2017. Springer International Publishing.

[DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[DF95] Rodney G Downey and Michael R Fellows. Fixed-parameter tractability and completeness II: On completeness for W[1]. *Theoretical Computer Science*, 141(1-2):109–131, 1995.

[DJW05] Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström. Counting models for 2SAT and 3SAT formulae. *Theoretical Computer Science*, 332(1):265–291, 2005.

[DO08] David Devlin and Barry O'Sullivan. Satisfiability as a Classification Problem. *Proc. of the 19th Irish Conf. on Artificial Intelligence and Cognitive Science*, 2008.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[DPVO23] Marco Dalla, Benjamin Provan-Bessell, Andrea Visentin, and Barry O'Sullivan. SAT Feature Analysis for Machine Learning Classification Tasks. In *Sixteenth International Symposium on Combinatorial Search, SOCS 2023, July 14-16, 2023, Prague, Czech Republic*, pages 138–142. AAAI Press, 2023.

[Dun24] Raphael Dunkel. One solver to rule all feature models—Or not?: Addressing the algorithm selection problem for# SAT. 2024.

[DVO21] Marco Dalla, Andrea Visentin, and Barry O'Sullivan. Automated SAT problem feature extraction using convolutional autoencoders. In *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 232–239. IEEE, 2021.

[DVO24] Marco Dalla, Andrea Visentin, and Barry O'Sullivan. A Machine Learning Approach to Model Counting. In *The 36th IEEE International Conference on Tools with Artificial Intelligence*. IEEE, 2024.

[Dwo08] Cynthia Dwork. Differential Privacy: A Survey of Results. *Proceedings of the 5th International Conference on Theory and Applications of Models of Computation*, pages 1–19, 2008.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing*, pages 61–75, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[EGS12] Stefano Ermon, Carla Gomes, and Bart Selman. Uniform Solution Sampling Using a Constraint Solver as an Oracle. In *Proceedings of UAI*, page 255–264, 2012.

[EO22] Guillaume Escamocher and Barry O'Sullivan. Generation and Prediction of Difficult Model Counting Instances. *CoRR*, abs/2212.02893, 2022.

[EOP20] Guillaume Escamocher, Barry O'Sullivan, and Steven David Prestwich. Generating difficult CNF instances in unexplored constrainedness regions. *Journal of Experimental Algorithmics (JEA)*, 25:1–12, 2020.

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In

Theory and Applications of Satisfiability Testing, pages 502–518. Springer, 2004.

[FH20] ABKFM Fleury and Maximilian Heisinger. CADICAL, KISSAT, PARACOOBA, PLINGELING and TREENGELING entering the SAT competition 2020. volume 2020, page 50, 2020.

[FHH21] Johannes K Fichte, Markus Hecher, and Florim Hamiti. The model counting competition 2020. *Journal of Experimental Algorithmics (JEA)*, 26:1–26, 2021.

[FHI+21] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT Competition 2020. *Artificial Intelligence*, 301:103572, 2021.

[FL19] Matthias Fey and Jan Eric Lenssen. Fast Graph Representation Learning with PyTorch Geometric. 2019.

[FLC+22] Thomas Fournier, Arnaud Lallouet, Télio Cropsal, Gaël Glorian, Alexandre Papadopoulos, Antoine Petitet, Guillaume Perez, Suruthy Sekar, and Wijnand Suijlen. A Deep Reinforcement Learning Heuristic for SAT based on Antagonist Graph Neural Networks. In *2022 IEEE 34th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1218–1222, 2022.

[Fur16] Steve Furber. Large-Scale Neuromorphic Computing Systems. *Journal of Neural Engineering*, 13(5):051001, 2016.

[GA20] Daniele Grattarola and Cesare Alippi. Graph Neural Networks in TensorFlow and Keras with Spektral, 2020.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[GBC+22] Maxime Gasse, Simon Bowly, Quentin Cappart, Jonas Charfreitag, Laurent Charlin, Didier Chételat, Antonia Chmiela, Justin Dumouchelle, Ambros Gleixner, Aleksandr M. Kazachkov, Elias Khalil, Pawel Lichocki, Andrea Lodi, Miles Lubin, Chris J. Maddison, Christopher Morris, Sebastian Pokutta, Antoine Prouvost, Lara Scavuzzo, Giulia Zarpellon, Linxin Yang, Sha Lai, Akang Wang, Xiaodong Luo, Xiang Zhou, Haohan Huang, Shengcheng Shao, Yuanming Zhu, Dong Zhang, Tao Quan, Zixuan Cao, Yang Xu, Zhewei Huang, Shuchang Zhou, Chen Binbin, He Minggui,

Hao Hao, Zhang Zhiyu, An Zhiwu, and Mao Kun. The Machine Learning for Combinatorial Optimization Competition (ML4CO): Results and Insights. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2022.

[GCL15] Jesús Giráldez-Cru and Jordi Levy. A modularity-based random SAT instances generator. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 1952–1958. AAAI Press, 2015.

[GCL21] Jesús Giráldez-Cru and Jordi Levy. Popularity-similarity random SAT formulas. *Artificial Intelligence*, 299:103537, 2021.

[GJ79] Michael R Garey and David S Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Company, 1979.

[GMGC22] Iván Garzón, Pablo Mesejo, and Jesús Giráldez-Cru. On the Performance of Deep Generative Models of Realistic SAT Instances. In *25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[GPAM+14] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, et al. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

[GSS06] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model Counting: A New Strategy for Obtaining Good Bounds. In *Proceedings of AAAI*, page 54–61, 2006.

[Gun17] David Gunning. Explainable Artificial Intelligence (XAI). *DARPA Broad Agency Announcement*, 2017.

[GW93] Ian P Gent and Toby Walsh. Towards an understanding of hill-climbing procedures for SAT. In *AAAI*, volume 93, pages 28–33. Citeseer, 1993.

[GZL+23] Wenxuan Guo, Hui-Ling Zhen, Xijun Li, Wanqian Luo, Mingxuan Yuan, Yaohui Jin, and Junchi Yan. Machine Learning Methods in Solving the Boolean Satisfiability Problem. *Machine Intelligence Research*, 20(5):640–655, June 2023.

[Han20] Jesse Michael Han. Enhancing SAT solvers with glue variable predictions, 2020.

[HD07] Jinbo Huang and Adnan Darwiche. The language of search. *Journal of Artificial Intelligence Research*, 29:191–219, 2007.

[HJ12] Cheng-Shen Han and Jie-Hong Roland Jiang. When Boolean Satisfiability Meets Gaussian Elimination in a Simplex Way. In *International Conference on Computer Aided Verification*, 2012.

[HJS09] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, June 2009.

[HKWB11] Marijn J.H. Heule, Oliver Kullmann, Sander Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Haifa Verification Conference*, pages 50–65. Springer, 2011.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Hol92] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, 1992.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[HS04] Holger H Hoos and Thomas Stützle. *Stochastic local search: Foundations and applications*. Elsevier, 2004.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2nd edition, 2009.

[HTH02] Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, pages 233–248, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[Hua07] Jinbo Huang. The effect of restarts on the efficiency of clause

learning. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, page 2318–2323, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[HvM09] Marijn J. H. Heule and Hans van Maaren. Look-Ahead Based SAT Solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 155–184. IOS Press, 2009.

[HXHLB14] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm Runtime Prediction: Methods & Evaluation. *Artificial Intelligence*, 206:79–111, 2014.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[IMM18] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python Toolkit for Prototyping with SAT Oracles. In *SAT*, pages 428–437, 2018.

[Jol02] Ian Jolliffe. Principal Component Analysis. *Springer Series in Statistics*, 2002.

[JP00] Roberto J. Bayardo Jr. and Joseph Daniel Pehoushek. Counting Models Using Connected Components. In *Proceedings of AAAI*, pages 157–162, 2000.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2017.

[KB23] Maximilian Kramer and Paul Boes. Using deep learning to construct stochastic local search SAT solvers with performance bounds. *arXiv preprint arXiv:2309.11452*, 2023.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[KGWC20] Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan Catanzaro. Can Q-Learning with Graph Networks Learn a Generalizable Branching Heuristic for a SAT Solver?, 2020.

[KLM89] Richard M Karp, Michael Luby, and Neal Madras. Monte-Carlo

approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429–448, 1989.

[KLW22] Tom Krüger, Jan-Hendrik Lorenz, and Florian Wörz. Too much information: Why CDCL solvers need to forget learned clauses. *PLOS ONE*, 17(8):e0272967, August 2022.

[KMS+11] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. *Algorithm Selection and Scheduling*, page 454–469. Springer Berlin Heidelberg, 2011.

[KPKT24] Bharti Khemani, Shruti Patil, Ketan Kotecha, and Sudeep Tanwar. A review of graph neural networks: concepts, architectures, techniques, challenges, datasets, applications, and future directions. *J. Big Data*, 11(1), January 2024.

[KSS08] Lukas Kroc, Ashish Sabharwal, and Bart Selman. Leveraging Belief Propagation, backtrack search, and statistics for model counting. *Proceedings of ISAIM*, 2008.

[KUMH17] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 972–981, Red Hook, NY, USA, 2017. Curran Associates Inc.

[KW13] Diederik P Kingma and Max Welling. Auto-Encoding Variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[KW14] Diederik P. Kingma and Max Welling. Semi-Supervised Learning with Deep Generative Models. In *Advances in Neural Information Processing Systems*, 2014.

[KW16] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 521(7553):436–444, 2015.

[LC21] Xue Li and Yuanzhi Cheng. Understanding the message passing in graph neural networks via power iteration clustering. *Neural Networks*, 140:130–135, 2021.

[LENV17] Massimo Lauria, Jan Elffers, Jakob Nordström, and Marc Vinyals. CNFgen: A generator of crafted benchmarks. In *Theory and Applications of Satisfiability Testing–SAT 2017: 20th International Conference, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings 20*, pages 464–473. Springer, 2017.

[LGPC16] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In Nadia Creignou and Daniel Le Berre, editors, *Proceedings of SAT*, pages 123–140, 2016.

[LH05] Chu-Min Li and Wen-Qin Huang. Diversification and determinism in local search for satisfiability. In *International conference on theory and applications of satisfiability testing*, pages 158–172. Springer, 2005.

[Lia18] Jia Liang. Machine Learning for SAT Solvers, 12/2018 2018.

[Lip18] Zachary C Lipton. The Mythos of Model Interpretability. *Communications of the ACM*, 61(10):36–43, 2018.

[LJN12] T. Laitinen, T. Junttila, and I. Niemelä. Conflict-Driven XOR-Clause Learning. *Journal of Automated Reasoning*, pages 61–75, 2012.

[LKP+17] Jia Hui Liang, Hari Govind V. K., Pascal Poupart, Krzysztof Czarnecki, and Vijay Ganesh. An Empirical Study of Branching Heuristics Through the Lens of Global Learning Rate. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2017.

[LL17] Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *CoRR*, abs/1705.07874, 2017.

[LLM16] Jean-Marie Lagniez, Emmanuel Lonca, and Pierre Marquis. Improving Model Counting by Leveraging Definability. In Subbarao

Kambhampati, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, 2016.

[LMSS16] Andrea Loreggia, Yuri Malitsky, Horst Samulowitz, and Vijay A. Saraswat. Deep Learning for Algorithm Portfolios. In *Proceedings of AAAI*, pages 1280–1286, 2016.

[LOG+18] Jia Hui Liang, Chee Yap Oh, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart. Learning Rate Based Branching Heuristic for SAT Solvers. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), 2018.

[LOM+18] Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh. Machine learning-based restart policy for CDCL SAT solvers. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 94–110, Cham, 2018. Springer International Publishing.

[LPPR19] Anastasia-Maria Leventi-Peetz, J.-V. Peetz, and Martina Rohde. ML Supported Predictions for SAT Solvers Performance. *ArXiv*, abs/2112.09438, 2019.

[LSL+22] Min Li, Zhengyuan Shi, Qiuxia Lai, Sadaf Khan, Shaowei Cai, and Qiang Xu. DeepSAT: An EDA-Driven Learning Framework for SAT. *arXiv preprint arXiv:2205.13745*, 2022.

[Mar21] Raffaele Marino. Learning from survey propagation: a neural network for MAX-E-3-SAT. *Machine Learning: Science and Technology*, 2(3):035032, 2021.

[MHB13] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. Automated reencoding of Boolean formulas. In Armin Biere, Amir Nahir, and Tanja Vos, editors, *Hardware and Software: Verification and Testing*, pages 102–117, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[ML98] Stephen M. Majercik and Michael L. Littman. Using Caching to Solve Larger Probabilistic Planning Problems. In *Proceedings of AAAI*, page 954–959, 1998.

[MM+01] Matthew W. Moskewicz, Conor F. Madigan, et al. Chaff: Engi-

neering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference (DAC)*, pages 530–535, 2001.

[MMBH12] Christian J. Muise, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu. Dsharp: Fast d-DNNF Compilation with sharpSAT. In Leila Kosseim and Diana Inkpen, editors, *Proceedings of the Canadian AI Conference*, volume 7310, pages 356–361. Springer, 2012.

[MML12] Ruben Martins, Vasco Manquinho, and Inês Lynce. An overview of parallel SAT solving. *Constraints*, 17, 07 2012.

[MMR+17] H Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, et al. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, pages 1273–1282, 2017.

[MR03] Daniel D. McCracken and Edwin D. Reilly. *Backus-Naur form (BNF)*, page 129–131. John Wiley and Sons Ltd., GBR, 2003.

[MSLM09] Joao P. Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 131–153. IOS Press, 2009.

[MSS99] Joao P Marques-Silva and Karem A Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[New03] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45(2):167–256, 2003.

[New06] M E J Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

[NKR+18] Nina Narodytska, Shiva Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. Verifying properties of binarized deep neural networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI*

Symposium on Educational Advances in Artificial Intelligence, AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018.

[NR12] Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 242–255. Springer, 2012.

[PC17] Gadi Pinkas and Shimon Cohen. Artificial Neural Networks that Learn to Satisfy Logic Constraints. *arXiv preprint arXiv:1712.03049*, 2017.

[PD07] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 294–299. Springer, 2007.

[PDVO22] Benjamin Provan-Bessell, Marco Dalla, Andrea Visentin, and Barry O'Sullivan. SATfeatPy - A Python-based Feature Extraction System for Satisfiability. *CoRR*, abs/2204.14116, 2022.

[Pea88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[Pre18] John Preskill. Quantum Computing in the NISQ Era and Beyond. *Quantum*, 2:79, 2018.

[Puk11] Llukan Puka. *Kendall's Tau*, pages 713–715. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[PY09] Sinno Jialin Pan and Qiang Yang. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2009.

[Qui86] J. Ross Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81–106, 1986.

[R+16] Marco Tulio Ribeiro et al. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144, 2016.

[RHVL20] Luca Rendsburg, Holger Heidrich, and Ulrike Von Luxburg. NetGAN without GAN: From random walks to low-rank approximations. In *International Conference on Machine Learning*, pages 8073–8082. PMLR, 2020.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[RN09] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

[RNSS18] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training. *arXiv preprint arXiv:1810.04805*, 2018.

[Rya04] Lawrence Ryan. Efficient algorithms for clause-learning SAT solvers. Simon Fraser University, 2004.

[S+16] David Silver et al. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 529(7587):484–489, 2016.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

[SB19] Daniel Selsam and Nikolaj Bjørner. *Guiding High-Performance SAT Solvers with Unsat-Core Predictions*, page 336–353. Springer International Publishing, 2019.

[SB22] Gaia Saveri and Luca Bortolussi. Graph Neural Networks for Propositional Model Counting. *arXiv preprint*, 2022.

[SBB+04] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. In *SAT*, 2004.

[SBK05] Tian Sang, Paul Beame, and Henry A. Kautz. Performing Bayesian Inference by Weighted Model Counting. In Manuela M. Veloso and Subbarao Kambhampati, editors, *The 20th AAAI Conference on Artificial Intelligence*, pages 475–482, 2005.

[SGBP20] Ling Sun, David Gérault, Adrien Benamira, and Thomas Peyrin. NeuroGIFT: Using a Machine Learning Based SAT Solver for Cryptanalysis. In *International Conference on Cryptology in Africa*. Springer, 2020.

[SGM19] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

[SKC94] Bart Selman, Henry A Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (vol. 1)*, pages 337–343. American Association for Artificial Intelligence, 1994.

[SLB+18] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT Solver from Single-Bit Supervision. *arXiv preprint arXiv:1802.03685*, 2018.

[SLB+21] Feng Shi, Chonghan Lee, Mohammad Khairul Bashar, Nikhil Shukla, Song-Chun Zhu, and Vijaykrishnan Narayanan. Transformer-based machine learning for fast SAT solvers and logic synthesis. *arXiv preprint arXiv:2107.07116*, 2021.

[SLK+22] Zhengyuan Shi, Min Li, Sadaf Khan, Hui-Ling Zhen, Mingxuan Yuan, and Qiang Xu. SAtformer: Transformers for SAT solving. *arXiv preprint arXiv:2209.00953*, 2022.

[SLM92] Bart Selman, Hector J Levesque, and David G Mitchell. A new method for solving hard satisfiability problems. In *AAAI*, volume 92, pages 440–446. Citeseer, 1992.

[Spe10] Ivor Spence. SGEN1: A generator of small but difficult satisfiability benchmarks. *Journal of Experimental Algorithmics (JEA)*, 15:1–1, 2010.

[SRSM19] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel.

GANAK: A Scalable Probabilistic Exact Model Counter. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1169–1176, 7 2019.

[Sto76] Larry J Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(1):237–267, 1976.

[SYZ+24] Yiwen Sun, Furong Ye, Xianyin Zhang, Shiyu Huang, Bingzhen Zhang, Ke Wei, and Shaowei Cai. Autosat: Automatically optimize SAT solvers via large language models. *CoRR*, abs/2402.10705, 2024.

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Thu06] Marc Thurley. sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP. volume 4121, pages 424–429, 07 2006.

[TL19] Mingxing Tan and Quoc Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning*, pages 6105–6114, 2019.

[Tod89] Seinosuke Toda. On the computational power of PP and (+)P. *30th Annual Symposium on Foundations of Computer Science*, pages 514–519, 1989.

[Tse83] Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pages 466–483, 1983.

[Tse21] TseKiChun. Random Forest Explain, 2021.

[Tur50] Alan M. Turing. Computing Machinery and Intelligence. *Mind*, 59(236):433–460, 1950.

[Tut59] WT Tutte. A theorem on planar graphs. *Proceedings of the Cambridge Philosophical Society*, 55:5–8, 1959.

[UONW23] Felix Ulrich-Oltean, Peter Nightingale, and James Alfred Walker. Learning to select SAT encodings for pseudo-Boolean and linear integer constraints. *arXiv*, 2023.

[vDDYS21] Ronald van Driel, Emir Demirović, and Neil Yorke-Smith. Learning variable activity initialisation for lazy clause generation solvers. In Peter J. Stuckey, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 62–71, Cham, 2021. Springer International Publishing.

[VRD09] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[VSP+17] Ashish Vaswani, Noam Shazeer, Niki Parmar, et al. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, 2017.

[WD92] Christopher J.C.H. Watkins and Peter Dayan. Q-Learning. In *Machine Learning*, volume 8, pages 279–292, 1992.

[Wes00] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2 edition, September 2000.

[WHT+21] Wenxi Wang, Yang Hu, Mohit Tiwari, Sarfraz Khurshid, Kenneth L. McMillan, and Risto Miikkulainen. NeuroComb: Improving SAT Solving with Graph Neural Networks. *CoRR*, abs/2110.14053, 2021.

[WHT+24] Wenxi Wang, Yang Hu, Mohit Tiwari, Sarfraz Khurshid, Kenneth McMillan, and Risto Miikkulainen. NeuroBack: Improving CDCL SAT Solving using Graph Neural Networks, 2024.

[WLY23] Fangzhou Wang, Jinwei Liu, and Evangeline F.Y. Young. FastPass: Fast Pin Access Analysis with Incremental SAT Solving. In *Proceedings of the 2023 International Symposium on Physical Design*, ISPD '23, page 9–16, New York, NY, USA, 2023. Association for Computing Machinery.

[WR18] Fei Wang and Tiark Rompf. From Gameplay to Symbolic Reasoning: Learning SAT Solver Heuristics in the Style of Alpha(Go) Zero. *arXiv preprint arXiv:1802.05340*, 2018.

[WR19] Haoze Wu and Raghuram Ramanujan. Learning to generate industrial SAT instances. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, pages 206–207, 2019.

[WS98] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

[WS05] Wei Wei and Bart Selman. A new approach to model counting. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing*, pages 324–339, Berlin, Heidelberg, 2005.

[Wu17] Haoze Wu. Improving SAT-solving with Machine Learning. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, page 787–788, 2017.

[WY23] Weihuang Wen and Tianshu Yu. W2SAT: Learning to generate SAT instances from Weighted Literal Incidence Graphs. *arXiv preprint arXiv:2302.00272*, 2023.

[WYC+19] Fei Wang, Zhanfu Yang, Ziliang Chen, Guannan Wei, and Tiark Rompf. Graph Neural Networks for Reasoning 2-Quantified Boolean Formulas. *arXiv preprint arXiv:1909.11830*, 2019.

[XHHLB08] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of artificial intelligence research*, 32:565–606, 2008.

[XHHLB12] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Evaluating Component Solver Contributions to Portfolio-Based Algorithm Selectors. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 228–241. Springer, 2012.

[XHLB10] Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pages 210–216, 2010.

[YCBL14] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How Transferable Are Features in Deep Neural Networks? In *Advances in Neural Information Processing Systems*, 2014.

[YLS+23] Zhiyuan Yan, Min Li, Zhengyuan Shi, Wenjie Zhang, Yingcong Chen, and Hongce Zhang. Addressing Variable Dependency in GNN-based SAT Solving. 2023.

[YLZ19] Wentan Yu, Huan Li, and Zhe Zhang. G2SAT: Learning to Gen-

erate SAT Formulas. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, pages 10552–10563, 2019.

[YP19] Emre Yolcu and Barnabás Póczos. Learning local search heuristics for boolean satisfiability. *Advances in Neural Information Processing Systems*, 32, 2019.

[YWB+19] Jiaxuan You, Haoze Wu, Clark Barrett, Raghuram Ramanujan, and Jure Leskovec. G2SAT: learning to generate SAT formulas. *Advances in neural information processing systems*, 32, 2019.

[YWC+19] Zhanfu Yang, Fei Wang, Ziliang Chen, Guannan Wei, and Tiark Rompf. Graph Neural Reasoning for 2-Quantified Boolean Formula Solvers. 2019.

[Zar05] Jerrold Zar. *Spearman Rank Correlation*, volume 5. 07 2005.

[ZCC+24] Zhanguang Zhang, Didier Chételat, Joseph Cotnareanu, Amur Ghose, Wenyi Xiao, Hui-Ling Zhen, Yingxue Zhang, Jianye Hao, Mark Coates, and Mingxuan Yuan. GraSS: Combining Graph Neural Networks with Expert Knowledge for SAT Solver Selection. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '24, page 6301–6311, New York, NY, USA, 2024. Association for Computing Machinery.

[ZMMM] Lintao Zhang, C.F. Madigan, M.H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, ICCAD-01, page 279–285. IEEE.

[ZSNL22] Kiarash Zahirnia, Oliver Schulte, Parmis Naddaf, and Ke Li. Micro and macro level graph modeling for graph variational auto-encoders. *Advances in Neural Information Processing Systems*, 35:30347–30361, 2022.

[ZSZ+20] Wenjie Zhang, Zeyu Sun, Qihao Zhu, Ge Li, Shaowei Cai, Yingfei Xiong, and Lu Zhang. NLocalSAT: Boosting Local Search with Solution Prediction. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, IJCAI-PRICAI-2020. International Joint Conferences on Artificial Intelligence Organization, July 2020.