SYNTHESIZING FEATURE EXTRACTORS: AN AGENTIC APPROACH FOR ALGORITHM SELECTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Feature engineering remains a critical bottleneck in machine learning, often requiring significant manual effort and domain expertise. While end-to-end deep learning models can automate this process by learning latent representations, they do so at the cost of interpretability. We propose a gray-box paradigm for automated feature engineering that leverages Large Language Models for program synthesis. Our framework treats the LLM as a meta-learner that, given a high-level problem description for constraint optimization, generates executable Python scripts that function as interpretable feature extractors. These scripts construct symbolic graph representations and calculate structural properties, combining the generative power of LLMs with the transparency of classical features. We validate our approach on algorithm selection across 227 combinatorial problem classes. Our synthesized feature extractors achieve 58.8% accuracy, significantly outperforming the 48.6 % of human-engineered extractors, establishing program synthesis as an effective approach to automating the ML pipeline.

1 INTRODUCTION

Understanding the structure of combinatorial optimization problems is fundamental to solving them efficiently. This structure manifests through various characteristics—the density, the modularity, the nature of their relationships, connectivity patterns, and the presence of known structural motifs. By identifying these characteristics as features, we can classify problems, predict their computational hardness, and most importantly, select the most suitable algorithmic approaches to solve them. Since the 1970s, algorithm selection has emerged as a cornerstone technique in machine learning (ML) and optimization, promising to match each problem instance with its most effective solver from a portfolio of complementary algorithms (Rice, 1976).

Yet despite decades of progress (Kerschke et al., 2019; Hoos et al., 2021; Kotthoff, 2016), a critical bottleneck remains: designing effective feature extractors. Creating a tool that compiles a problem instance into a succinct and representative set of features requires deep domain expertise, intimate knowledge of which features matter for algorithmic performance, and efficient extraction methods that do not consume excessive computational resources. This challenge becomes particularly acute when facing new problem domains. Research groups and companies encountering novel combinatorial problems find themselves unable to leverage powerful techniques like algorithm selection and algorithm configuration, simply because no feature extractor exists for their specific problem.

The conventional workaround—translating the problem into another formalism for which extractors exist—often proves inadequate. When we compile a high-level problem description into a flat constraint representation, we lose crucial structural information that was explicit in the original formulation. Global properties and graphical relationships that are immediately apparent in the high-level description become obscured or entirely invisible once the problem is flattened. This loss of information directly impacts the quality of algorithm selection, as the features extracted from the flat representation fail to capture characteristics that truly differentiate easy instances from hard ones.

Our Approach: Automating Feature Extraction via LLMs In this paper, we present a novel approach that fundamentally changes how feature extractors are created. Rather than requiring manual engineering by domain experts, we introduce an LLM-based framework that automatically generates

executable Python scripts serving as feature extractors. Our key insight is a *two-level process*. We first use a Large Language Model via an agentic error correcting workflow to generate an executable program. This program then serves as the feature extractor, capturing the essential characteristics of the problem.

Our approach operates on high-level problem descriptions written in MiniZinc (Stuckey et al., 2014; Marriott et al., 2008), a declarative modeling language for constraint satisfaction and optimization problems. This choice is deliberate: rich, expressive formalisms enable more compact problem representations, making it easier for the LLM to identify and exploit structural patterns. The LLM agent analyzes the MiniZinc model and generates a complete Python script that, when executed on a problem instance, produces a graph representation of the problem instance, which in turn gives rise to a vector of interpretable features.

Critically, our framework produces explicit, interpretable features rather than opaque neural embeddings. While recent work has explored using deep learning to create latent representations of problems (Pellegrino et al., 2025; Zhang et al., 2024; Loreggia et al., 2016), such approaches sacrifice transparency for automation. In contrast, our generated extractors produce features that domain experts can understand, validate, and refine: characteristics like graph density, variable clustering coefficients, constraint tightness, and statistical properties of the data. This "gray box" approach ensures that the automated extraction process remains accessible to human understanding and improvement.

Two Complementary Frameworks We develop two distinct but complementary frameworks for feature extraction.

The *problem-specific framework* generates tailored extractors for individual problem types, analyzing the particular structure and semantics of problems like *vehicle routing problem* (VRP) or *car sequencing* (CS). In this framework, the LLM agent is provided with the high-level problem description, instance data, and schema information. From these inputs, it generates a specialized Python script that extracts, via the construction of a custom graphical representation of the instance, approximately 50 characteristics relevant to algorithm selection for that particular problem.

The problem-generic framework takes a more ambitious approach: applying a universal feature extractor to *any* constraint satisfaction problem. The key innovation here is the use of a *universal* intermediate graph representation (in contrast to a custom one in the problem-specific framework). The LLM-generated script first converts any problem instance into a standardized bipartite graph, with nodes representing variables, constraints, and resources, and weighted edges encoding their relationships. From this graph representation, we extract a uniform set of structural features using standard graph analysis algorithms. This two-level approach, where the LLM generates a script that converts the problem into a graph, from which features are then extracted, preserves high-level structural information while enabling consistent feature extraction across diverse problem types.

Empirical Validation and Surprising Results To validate our approach, we conducted experiments in algorithm selection scenarios using a portfolio of five leading solvers: Gurobi, CPLEX, SCIP, Gecode, and OR-Tools. For the problem-specific framework, we evaluated performance on three distinct problem types with sufficient instance diversity. For the problem-generic framework, we used a benchmark of over 2000 instances spanning 227 different problems from two decades of MiniZinc Challenges (2008-2025).

Our automatically-generated extractors consistently and substantially outperform *mzn2feat*, the established feature extractor for MiniZinc problems (Amadini et al., 2013; 2014). Algorithm selectors trained on our features achieved 58.8% accuracy on the generic benchmark suite, compared to 48.6% for those using *mzn2feat* features, a notable improvement that held across different ML models and problem types. This performance gain stems from our extractors' ability to capture high-level structural properties that remain hidden in flat representations.

Contributions and Significance This work makes several significant contributions to combinatorial optimization and algorithm selection:

1. We demonstrate that LLMs can effectively reason about combinatorial problem structure and generate functional feature extractors wth minimal human guidance. This automation

democratizes access to advanced algorithm selection techniques, enabling their application to new problem domains where manual feature engineering would be prohibitively expensive.

2. We introduce the novel approach of using LLM-generated scripts as an intermediate representation, preserving interpretability while achieving automation. Unlike end-to-end neural approaches, our framework produces extractors that humans can understand, validate, and improve upon, facilitating human-AI collaboration in algorithm design.
3. Through comprehensive empirical evaluation, we show that automatically generated extractors can surpass carefully engineered alternatives, suggesting that LLMs can identify subtle structural patterns that human experts might overlook. This finding has immediate practical implications for building better algorithm portfolios and improving the efficiency of combinatorial problem solving.
4. Our framework operates within reasonable computational budgets, making it accessible to researchers and practitioners without extensive resources. The generated extractors themselves are lightweight, adding minimal overhead to the algorithm selection pipeline.

Paper Organization The paper is organized as follows: we first introduce preliminaries for our problems, and then present the two frameworks. Next, we discuss the experimental results and analyses; finally, we conclude the paper and give promising further directions.

2 PRELIMINARIES

Definition 1 *The* Algorithm Selection Problem *(AS) (Kerschke et al., 2019)) is the optimization problem which accepts as input a portfolio $\mathcal{P}$ of algorithms $A \in \mathcal{P}$, a set of instances I, a Performance Metric $PM(A, I)$ which measures the performance of A on the set of instances I, and an execution budget B (time limits, memory limits, etc). The objective of the optimization is to find, without exceeding the resources indicated by B, an algorithm $A \in \mathcal{P}$ that maximizes the expected performance of A over the set of instances I. Note that in the whole paper, we use instance to refer to a problem instance instead of a sample point in the ML domain.*

Definition 2 *The* Single Best Solver *(SB) is the algorithm $A^{SB} \in \mathcal{P}$ that achieves the best overall performance with respect to the performance metric PM across the entire set of instances I, i.e.,*

$$A^{SB} = \arg\max_{A \in \mathcal{P}} PM(A, I).$$

The Single Best Solver strategy corresponds to selecting the Single Best Solver for all instances.

Definition 3 *The* Virtual Best Solver *(VBS) is the (hypothetical) per-instance selector that, for each instance $i \in I$, chooses the algorithm $A \in \mathcal{P}$ that achieves the best performance on that instance with respect to PM. Formally, its performance is defined as*

$$PM(\textit{VBS}, I) = \sum_{i \in I} \max_{A \in \mathcal{P}} PM(A, \{i\}).$$

The VBS serves as an upper bound on the achievable performance of any AS strategy, assuming perfect knowledge of per-instance performance.

Definition 4 *A Large Language Model agent,* ***LLM-agent*** *(Yao et al., 2023) is a tuple $\mathcal{A} = (L, T, M, \pi, E)$, where L is a large language model used for reasoning and generation, T is a set of external tools or APIs accessible by the agent, M is the memory module (short-term and/or long-term), π is the prompting policy that maps observations and history to model inputs, E is the environment with which the agent interacts via observations and actions.*

The LLM-agent agent operates in a loop: $o_t \xrightarrow{\pi} p_t \xrightarrow{L} a_t \xrightarrow{T,E} o_{t+1}$ where o_t is the observation at time t, p_t is the constructed prompt, a_t is the action (e.g., tool call, output), and o_{t+1} is the new observation.

Definition 5 *MiniZinc (Nethercote et al., 2007) is a high-level, declarative modeling language used to describe constraint satisfaction and optimization problems in a solver-independent way.*

A MiniZinc model file (.mzn) defines the problem: it includes variables, constraints, and optionally an objective (e.g., to minimize or maximize something).

A MiniZinc data file (.dzn) provides input data for the model, i.e., specific values for parameters declared in the model file.

3 ALGORITHM FRAMEWORKS

In this section, we present the frameworks for our two LLM-based agents, which are inspired by the recent agentic framework bridging LLM and CSP solving (Szeider, 2025). Our LLM-agent system takes a problem instance (including a .mzn and a .dzn file) in the MiniZinc language as input and outputs a Python script that extracts relevant features from the input problem instances.

3.1 PROBLEM-SPECIFIC LLM-BASED AGENT

For the specific-problem framework, the inputs of the agent problem-related files (minizinc model file .mzn and problem instance file .dzn) and a data schema that can provide the LLM with a detailed understanding of the input data structure. Meanwhile, we have two prompts: (script_system_prompt and mzn-tuning) controlling the generation of feature extractors (a Python program) in an efficient way.

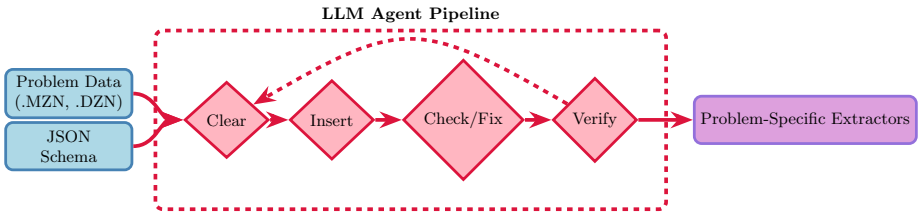


Figure 1: The workflow of generating problem-specific feature extractors

script_system_prompt is a general-purpose Python script providing behavioral instructions for LLM agents to generate complete, executable Python scripts through a tool-based environment. It defines a strict workflow, technical requirements, and available resources. The example of the general Python script template structure is in the appendix. The general-purpose script generation prompt follows a structured tool-based workflow, and its procedure follows Figure 1:

1. **Clear**: Clearing all previous content in the Python script.
2. **Insert**: Creating complete Python script.
3. **Check/Fix**: Verifying the syntax and requirements from the prompts; Addressing validation errors if needed.
4. **Execute**: Running the Python script and capturing and validating the output

The mzn-tuning prompt contains specialized instructions for LLM agents to generate constraint optimization feature extraction scripts. Unlike the general script system prompt, this is highly domain-specific for AS, guiding the agent to generate Python scripts that extract standardized instance features from constraint programming problems to train algorithm selectors for optimal solvers. The feature extractor template specifically for the Minizinc instance is in the appendix. The specialized prompt follows a research-oriented feature extraction workflow:

1. **Template Substitution**: Replace placeholders `${INSTANCE}` (.dzn file), `${SCHEMA}` (the prompt helping LLM understand the .mzn model file), `${MODEL}` (.mzn file), `${PROBLEM}` (problem name).
2. **Mandatory Imports**: Exact import requirements for framework integration
3. **Data Access**: Use `input_data()` - no file I/O operations

4. **Feature Analysis**: Extract 50 standardized features
5. **Testing**: `execute_script()` validation required
6. **Output**: Standardized format for algorithm selector training

3.2 Problem-generic LLM-based agent

In a further step, we design a generic workflow working on multiple cross-domain problems at the same time. The framework is a generic problem-generic pipeline where the output Python scripts include not only a parser and a feature extractor, like the problem-specific pipeline, but also a generic graph builder. After the parser handles and processes the input data, the generic graph builder constructs a graph with different types of nodes (like variables, constraints, and so on), and weighted edges representing their relationships. Based on the generic graph, the feature extractor applies graph analysis to extract structural features in a uniform way, obtaining the same features for diverse CP problems. The Figure 2 shows a monolithic architecture of the problem-generic pipeline for generating feature extractors. It has a similar overall framework to the problem-specific one, but we use a subagent to generate Python scripts (converters), integrating graph builders.

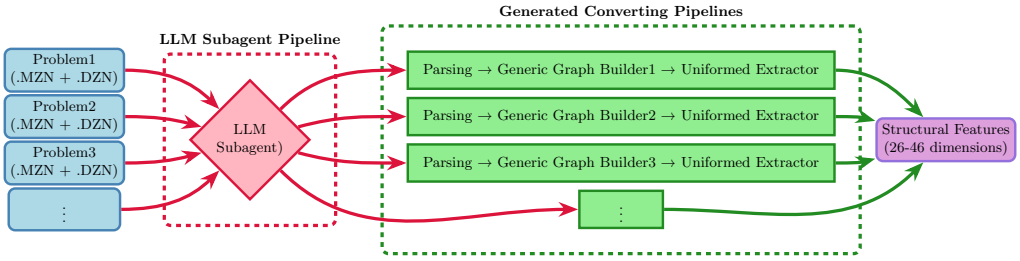


Figure 2: The workflow of generating problem-generic feature extractors

4 Experimental Analyses

4.1 Experimental Settings

In an AS problem, the goal is to select the most suitable (best) solver from a pool of candidates to solve a given MiniZinc instance, which consists of a model file and a corresponding data file. The pool $\mathcal{P}$ of solvers is composed of the best performing solvers in the Minizinc challenge [1]: Gurobi (12.0.3), CPLEX (22.1.2), SCIP (9.2.3), Gecode (6.2.0), and OR-Tools (9.3.10497). The set of instances I includes minimization/maximization problems, where the best solver means achieving the lowest/highest objective values after the timeout, and decision problems, where the best solver means the shortest solving time to get the results. In particular, for the experiments of the problem-specific framework, we extract features and train algorithm selectors on three problems: VRP (Queiroga et al., 2021), CS (Pellegrino et al., 2025), and fixed-length error correcting codes (FLECC) (Pellegrino et al., 2025), where we can get enough instances for training and testing fairly. In the experiments of the problem-generic framework, we comprehensively collect all MiniZinc Challenges (Stuckey et al., 2014) [2] from 2008 to 2025, where we have more than 2000 instances spanning 227 problems of combinatorial optimization after filtering out the problems with fewer than three instances. For both scenarios, problem-specific and problem-generic frameworks, we split the instances randomly into train and test sets by $7:3$ ratio. To keep each pair of comparisons fair, we use the same split of training and test sets.

We run solvers with a 20 mintues timeout to get the evaluation. We run the LLM pipeline to generate with *LLM2feat* extractors with a 1 minute timeout. We run all the instances on all solvers on a compute cluster with nodes equipped with two AMD 7403 processors (24 cores at 2.8 GHz) and 32 GB of RAM per core. The performance metric is $PM(A, I)$ where A is the solver and I is the set of instances, and $PM(A, I)$ is the number of instances for which solver A is the best solver.

[1] https://www.minizinc.org/challenge/2025/results/
[2] https://www.minizinc.org/challenge/

Accordingly, we have two metrics: the ratio of selecting the best solver (Acc), and the average ranking of selected solvers ($Rank$) for problem-specific evaluation. Additional Borda score from the competition [3] is used for cross-problem evaluation on the problem-generic framework. It generally means how many other solvers a solver can outperform.

For the selection of LLM models, we use OpenAI o4-mini-2025-04-16 and Anthropic claude-sonnet-4-20250514 in the problem-specific and problem-generic framework, respectively. Details about the LLM model sensitivity and selection refer to the Appendix.

4.2 Related Tools for Algorithm Selection

The input to the AS tools includes a feature table where there are problem instances with their corresponding features generated by feature extractors, and a performance table, which includes the instances and different solvers' performance metrics when solving the instances. To get the features of the instances we use as feature extractors, our LLM-based frameworks (*LLM2feat*) and *mzn2feat*.

In particular, we use the following AS tools in our experimental analysis:

Multiclass classification models for predicting the best solver. We use Random Forest (referred to as RF) in the standard way (Kerschke et al., 2019), and AutoSklearn (Feurer et al., 2015; 2020) (referred to as AutoSK). Their corresponding parameter settings are listed in the Appendix. We use two loss functions: accuracy, Acc, and average ranking, $Rank$, during training. By accuracy, we mean the ratio of times the model predicts the actual best solver, and by average ranking, we mean the average rank of the solver the model predicts. For all training, we use the cross-validation with 5 folds.

AutoFolio (Lindauer et al., 2015) (referred to as AF) and LLAMMA (Kotthoff, 2013) (referred to as LLAMMA), which are the best performing AS tools as reported in surveys (Kerschke et al., 2019) and competition [4]. For these tools, only the accuracy loss function is used for training, because we just use the tools in their default settings.

The combination of the above AS tools and feature extractors yields the following list of AS approaches: *mzn2feat*+AF; *mzn2feat*+RF; *mzn2feat*+LLAMMA; *mzn2feat*+AutoSK; *LLM2feat*+AF; *LLM2feat*+RF; *LLM2feat*+LLAMMA; *LLM2feat*+AutoSK.

4.3 Goals of the Experimental Analyses

We want to answer the following questions by extensive experiments:

- Q1: How diverse are features generated by *LLM2feat*? Ideal features in the same set are expected to be informatively diverse. Highly correlated features provide overlapping information, reducing the effective dimensionality of the feature space. Diverse features enable ML models to possibly capture complementary problem features and different aspects of problem complexity with fewer parameters.
- Q2: How efficiently does each feature contribute to the algorithm selection models? By analyzing the importance of features, we can generally understand the quality of the features and know the potential for reducing the redundant feature sets.
- Q3: How accurate are the AS models using *LLM2feat* and *mzn2feat* features? Applying the feature sets to AS models can directly validate how useful the features are for AS.

4.4 problem-specific feature extractor

For our problem-specific feature extractors, on the three different problems, we analyze feature sets extracted by *mzn2feat* and *LLM2feat* extractors by an extensive series of experiments. Firstly, we check the correlation between the features in both sets and the general diversity of the feature sets; then we check the features' importance and quality of the features generated by *mzn2feat* and our *LLM2feat*; Finally, we verify the effectiveness by training algorithm selectors with *mzn2feat*-based and *LLM2feat*-based features and compare their corresponding accuracy.

[3] https://www.minizinc.org/challenge/2025/

[4] https://www.coseal.net/open-algorithm-selection-challenge-2017-oasc/

4.4.1 Feature Correlation Analysis

Given a feature set, we can train a standard algorithm selector using random forests as described in 4.1. Consequently, we obtain two algorithm selectors, *mzn2feat* + RF and *LLM2feat* + RF, from which we can get the top 20 features of the node that contribute the most to the classification decision. This approach ensures our correlation analysis focuses on algorithm-selection relevant features rather than examining all features indiscriminately. For each feature set, we computed Pearson correlation matrices (Guyon & Elisseeff, 2003) and analyzed both the distribution of correlation coefficients (r) and their statistical properties. We show the heatmap figures of correlation coefficients from two feature sets on VRP problems (The results of FLECC and CS problems are in the subsection of the Appendix). LLM features show the most substantial diversity advantage with an average correlation magnitude of $|r| = 0.221$ compared to *mzn2feat*'s $|r| = 0.429$. For the other two problems, the tendency is also the same: these results demonstrate that LLM-generated features capture more diverse aspects of problem structure, reducing information redundancy and enabling more efficient representation of constraint optimization characteristics with fewer overlapping features.

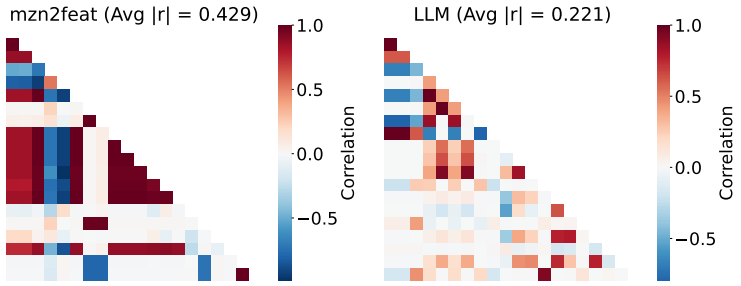
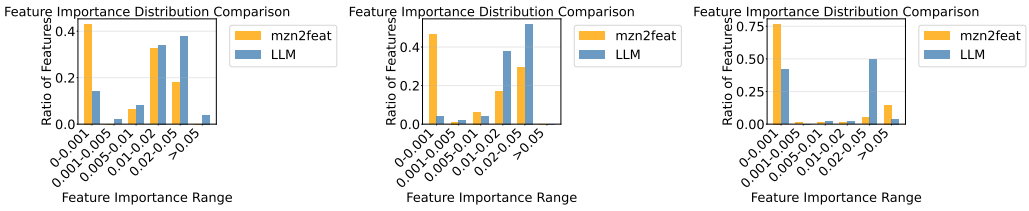


Figure 3: Feature correlation analysis for VRP problem (feature names removed for clarity). LLM features exhibit the largest diversity advantage with 48.5% lower average correlation ($|r| = 0.221$) compared to *mzn2feat* ($|r| = 0.429$).

4.4.2 Feature Utilization Efficiency

Beyond correlation analysis, we analyze how important each feature is in feature sets. This analysis addresses Q2 by evaluating the distribution of feature importance across all available features in each dataset.



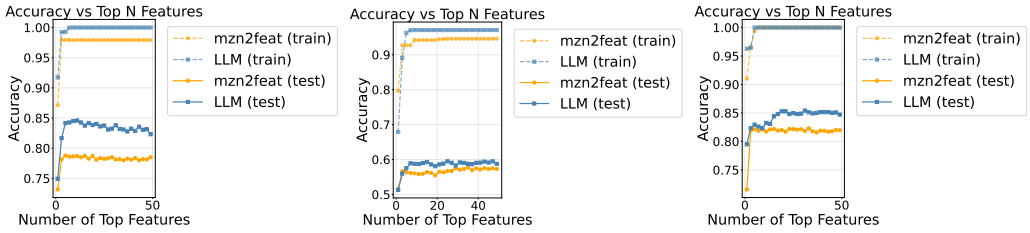
(a) FLECC Problem (b) CS Problem (c) VRP Problem

Figure 4: Feature importance distribution analysis across three constraint optimization problems. Consistent patterns show LLM features achieve better distribution across importance ranges.

We analyzed feature utilization efficiency using a comprehensive approach that examines the complete feature set rather than focusing solely on top-performing features. For both *mzn2feat*+RF and *LLM2feat*+RF, we extracted feature importance scores for all features and applied a significance threshold of 0.001 to identify "effectively utilized" features. This threshold ensures we capture features that contribute meaningfully to algorithm selection decisions while filtering out noise. Our analysis encompassed all 95 human-crafted features of *mzn2feat* and 50 *LLM2feat*-based features. The problem-specific framework produced features that demonstrate consistently superior utilization efficiency across all three problems. From Figure 4, for example, on the VRP problem, LLM achieves 96% utilization efficiency compared to *mzn2feat*'s 56.8%, representing a 69% improvement in effective feature usage. On FLECC, LLM reaches 58% efficiency versus *mzn2feat*'s 23.2%; On CS problem, LLM attains 84% efficiency compared to *mzn2feat*'s 45.1%.

4.4.3 ACCURACY ANALYSIS

To directly evaluate the practical impact of our feature extraction approaches, we conducted a comprehensive accuracy analysis that addresses Q3. This analysis examines algorithm selection performance as a function of feature set size, providing insights into both the effectiveness and efficiency of LLM-generated versus traditional features.



(a) FLECC Problem (b) CS Problem (c) VRP Problem

Figure 5: Accuracy analysis across three constraint optimization problems demonstrating consistent LLM superiority in algorithm selection performance and feature efficiency.

We implemented a systematic feature scaling analysis that evaluates algorithm selection accuracy using incrementally expanding feature sets. Starting with the single most important feature from each extraction method, we progressively added features in the order of decreasing importance, testing every other feature count from 1 to 50 features for computational efficiency. For each feature subset size, we selected $top - N$ features based on Random Forest feature importance rankings, and retrained algorithm selectors using only the selected features. Finally, we evaluated performance on both training and testing sets. From Figure 6, we can see LLM-based features reach near-optimal performance with greater accuracy with different numbers of features. Meanwhile, *LLM2feat*-based algorithm selectors have higher chances to improve the accuracy when more features are given, like on the VRP problem, when there are more than 10 features, *mzn2feat*-based features do not help for further improvement (the accuracy is always around 81%), but *LLM2feat*-based features can reach the highest values when more than 20 features are given.

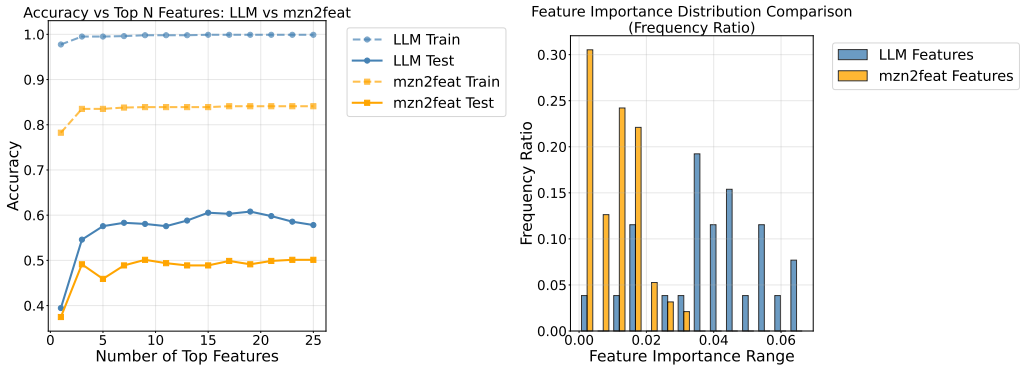
To verify that *LLM2feat* provides more informative structural features than *mzn2feat*, we also use respective features for training the algorithm selector with other ML tools (AF, LLAMMA, and AutoSK) besides RF for the three problems. From Table 1, we can see that, except when features are fed to AF, which always gets the same results as SB, the algorithm selectors based on *LLM2feat* always get higher accuracy and better ranking (lower ranking means better) on the test set. Therefore, *LLM2feat* can always get more informative problem feature sets for ML tools to utilize to select better solvers. In the Appendix, we also have the results on the training set with Acc as the loss function, and the results on the training and test sets with $Rank$ as the loss function.

Loss function=Acc (Testing)	VRP		CS		FLECC		Suite (227 problems)		
	Acc	$Rank$	Acc	$Rank$	Acc	$Rank$	Acc	$Rank$	Borda
SB	79.5%	1.221	49.0%	**1.616**	78.2%	1.420	27.0%	2.667	0.694
mzn2feat+AF	79.5%	1.221	49.0%	**1.616**	78.2%	1.420	27.0%	2.667	0.694
mzn2feat+RF	83.0%	1.184	58.5%	1.680	79.5%	1.391	47.4%	1.419	1.323
mzn2feat+LLAMMA	82.9%	1.184	59.4%	1.798	78.7%	1.397	×	×	×
mzn2feat+AutoSK	84.4%	1.166	62.1%	1.787	79.4%	1.394	48.6%	1.464	1.320
LLM2feat+AF	79.5%	1.221	49.0%	**1.616**	78.2%	1.420	27.0%	2.667	0.694
LLM2feat+RF	85.7%	**1.155**	61.3%	1.681	83.5%	1.338	58.3%	**1.340**	**1.426**
LLM2feat+LLAMMA	85.8%	1.157	61.0%	1.773	84.2%	**1.306**	×	×	×
LLM2feat+AutoSK	**85.4%**	1.160	**64.0%**	1.738	**84.6%**	1.310	**58.8%**	1.390	1.421

Table 1: The accuracy and the average ranking results of *mzn2feat* features (95) and *LLM2feat* features (50) trained with different models on the testing set (loss function= Acc). Note: the results of the problem-generic framework on the problem suite, including 227 problems, are also listed here, and they will be discussed later. LLAMMA is not applicable to cross-problem algorithm selection, so it is masked.

4.5 PROBLEM-GENERIC FEATURE EXTRACTOR

For the more general framework, the problem-generic framework, we test them in a more systematic way on 227 problems. We also analyze the feature correlation, feature utilization efficiency, and accuracy analysis as we do for the problem-specific framework. The outperformance of the features generated by *LLM2feat* is even more significant compared with that generated by *mzn2feat*.



(a) Accuracy with Top Features (b) Frequency ratio in different importance range

Figure 6: Accuracy analysis across 227 constraint optimization problems demonstrating consistent LLM superiority in algorithm selection performance, and informative features obtained by LLM than *mzn2feat*

For feature correlation, LLM features show significantly lower average correlation ($|r| = 0.141$) compared to *mzn2feat* ($|r| = 0.245$), demonstrating the improvement in feature diversity (The heapmap figure is in the Appendix). From the feature importance distribution comparison in Figure 6, the feature utilization of LLM-based features is more efficient as *LLM2feat* generates more features with higher importance than *mzn2feat*. On the test accuracy with the top important features, the *LLM2feat*-based RF always gets higher accuracy than the *mzn2feat*-based RF, which means given the same number of important features from respective feature sets, *LLM2feat*-based features are more informative for training algorithm selectors. For example, with around 15 top features, *LLm2feat*-based RF can get 60% on the test set, which is much higher than the accuracy (49%) obtained by *mzn2feat*-based RF.

From Table 1, we have the results of different metrics obtained by various *LLM2feat* and *mzn2feat*-based algorithm selectors: LLM-based algorithm selectors can achieve at least 10% higher accuracy, and better ranking. Moreover, we add another evaluative metric, Borda scores, which also indicates *LLM2feat*-based algorithm selectors can also get higher scores than their competitors.

5 CONCLUSION AND FUTURE WORK

To the best of our knowledge, we have introduced the first framework for the automatic construction of feature extractors for combinatorial problems. The framework is grounded in the use of LLM agents, which are parametrically instructed to accommodate potentially diverse representation formalisms. The extractors are generated as Python scripts that can subsequently be refined by human experts in a "gray-box" manner. The construction process is computationally efficient and can therefore be employed in an on-demand fashion for dynamic scenarios. Moreover, we have demonstrated the effectiveness of these extractors in a representative ML task—algorithm selection—showing that they can achieve competitive performance even in domains where human-engineered feature extractors are already available.

REPRODUCIBILITY STATEMENT

We took several steps to make our results easy to reproduce.

Code and artifacts. We release an anonymized supplement for review containing: (i) the full training/evaluation pipeline for all selectors, (ii) the LLM prompting templates and the exact prompts used, (iii) all LLM-generated feature-extractor scripts (with content hashes).

Data and benchmarks. All experiments use public MiniZinc Challenge benchmarks (2008–2025). We provide scripts to download the models and instances, and a manifest listing the subset used after filtering problems with fewer than 3 instances. We include our fixed train/test split (70/30) as instance lists to ensure identical data partitions across runs.

Solvers and versions. We report and pin solver versions in all runs: Gurobi 12.0.3, CPLEX 22.1.2, SCIP 9.2.3, Gecode 6.2.0, OR-Tools 9.3.10497. We provide wrapper scripts that uniformly handle time limits, seeds, and logging. Proprietary solvers (Gurobi/CPLEX) are optional; our scripts fall back to open-source solvers (SCIP, Gecode, OR-Tools).

LLM configurations. For problem-specific extractors we used OpenAI `o4-mini-2025-04-16`; for the problem-generic pipeline we used Anthropic `claude-sonnet-4-20250514`. We release: (a) the system and task prompts, (b) the template files, (c) the top-level orchestration script, and (d) the raw generated extractors. We cap extractor generation to 60 s per script and keep all retries; each artifact is identified by a SHA-256 hash.

Learning pipelines and hyperparameters. We evaluate Random Forest (RF), AutoSklearn (AutoSK), AutoFolio (AF), and LLAMA with default settings unless specified.

Evaluation protocol. For each instance we run all portfolio solvers with a 20-minute wall-clock limit and parse outcomes into a unified performance table. Selector models are trained on the training split only; metrics are reported on the fixed test split. We report: (i) selector accuracy (fraction of instances where the predicted solver is best), (ii) average rank (lower is better), and for the cross-problem suite also the MiniZinc Borda score. Scripts regenerate all tables/plots from logs.

Hardware. Experiments were run on a cluster with nodes equipped with two AMD 7403 processors (24 cores @ 2.8 GHz) and 32 GB RAM per core. We provide the job scripts used on our scheduler; the pipeline also runs on a single workstation (with longer runtimes) using the provided Docker image.

Determinism and seeds. Where components are nondeterministic, we set seeds (42) and document any remaining sources of variability (e.g., parallel tree building, AutoSklearn's ensembling). All reported numbers are from a single, fixed split; we include scripts to rerun with new splits and to aggregate across folds if desired.

Availability. The anonymized reproduction package includes: code, configs, prompts, generated extractors, and figure/table notebooks.

REFERENCES

Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Features for building CSP portfolio solvers. *CoRR*, abs/1308.0227, 2013. URL http://arxiv.org/abs/1308.0227.

Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An enhanced features extractor for a portfolio of constraint solvers. In Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong (eds.), *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, pp. 1357–1359. ACM, 2014. doi: 10.1145/2554850.2555114. URL https://doi.org/10.1145/2554850.2555114.

Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems 28 (2015)*, pp. 2962–2970, 2015.

Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Auto-sklearn 2.0: Hands-free automl via meta-learning. *arXiv:2007.04074 [cs.LG]*, 2020.

Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3(null):1157–1182, March 2003. ISSN 1532-4435.

Holger H Hoos, Frank Hutter, and Kevin Leyton-Brown. Automated configuration and selection of sat solvers. In *Handbook of Satisfiability*, pp. 481–507. IOS Press, 2021.

Pascal Kerschke, Holger H. Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evol. Comput.*, 27(1):3–45, 2019. doi: 10.1162/EVCO_A_00242. URL https://doi.org/10.1162/evco_a_00242.

Lars Kotthoff. LLAMA: leveraging learning to automatically manage algorithms. Technical Report arXiv:1306.1031, arXiv, June 2013. URL http://arxiv.org/abs/1306.1031.

Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. In *Data mining and constraint programming: Foundations of a cross-disciplinary approach*, pp. 149–190. Springer, 2016.

M. Lindauer, H. Hoos, F. Hutter, and T. Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015.

Andrea Loreggia, Yuri Malitsky, Horst Samulowitz, and Vijay A. Saraswat. Deep learning for algorithm portfolios. In Dale Schuurmans and Michael P. Wellman (eds.), *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pp. 1280–1286. AAAI Press, 2016. doi: 10.1609/AAAI.V30I1.10170. URL https://doi.org/10.1609/aaai.v30i1.10170.

Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the zinc modelling language. *Constraints An Int. J.*, 13(3): 229–267, 2008. doi: 10.1007/S10601-008-9041-4. URL https://doi.org/10.1007/s10601-008-9041-4.

Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere (ed.), *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pp. 529–543. Springer, 2007. doi: 10.1007/978-3-540-74970-7_38. URL https://doi.org/10.1007/978-3-540-74970-7_38.

Alessio Pellegrino, Özgür Akgün, Nguyen Dang, Zeynep Kiziltan, and Ian Miguel. Transformer-Based Feature Learning for Algorithm Selection in Combinatorial Optimisation. In Maria Garcia de la Banda (ed.), *31st International Conference on Principles and Practice of Constraint Programming (CP 2025)*, volume 340 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 31:1–31:22, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-380-5. doi: 10.4230/LIPIcs.CP.2025.31. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CP.2025.31.

Eduardo Queiroga, Ruslan Sadykov, Eduardo Uchoa, and Thibaut Vidal. 10,000 optimal cvrp solutions for testing machine learning based heuristics. In *AAAI-22 workshop on machine learning for operations research (ML4OR)*, 2021.

John R. Rice. The algorithm selection problem**this work was partially supported by the national science foundation through grant gp-32940x. this chapter was presented as the george e. forsythe memorial lecture at the computer science conference, february 19, 1975, washington, d. c. volume 15 of *Advances in Computers*, pp. 65–118. Elsevier, 1976. doi: https://doi.org/10.1016/S0065-2458(08)60520-3. URL https://www.sciencedirect.com/science/article/pii/S0065245808605203.

Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The minizinc challenge 2008–2013. *AI Magazine*, 35(2):55–60, Jun. 2014. doi: 10.1609/aimag.v35i2.2539. URL https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2539.

Stefan Szeider. Bridging language models and symbolic solvers via the model context protocol. In Jeremias Berg and Jakob Nordström (eds.), *28th International Conference on Theory and Applications of Satisfiability Testing, SAT 2025, August 12-15, 2025, Glasgow, Scotland*, volume 341 of *LIPIcs*, pp. 30:1–30:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi: 10.4230/LIPICS.SAT.2025.30. URL https://doi.org/10.4230/LIPIcs.SAT.2025.30.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

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 Ricardo Baeza-Yates and Francesco Bonchi (eds.), *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25-29, 2024*, pp. 6301–6311. ACM, 2024. doi: 10.1145/3637528.3671627. URL https://doi.org/10.1145/3637528.3671627.

A Appendix

A.1 General Python Script Template guided by script-system-prompt.md

```python
# Required script structure
import necessary_modules
# Constants and configuration
CONSTANTS = values
# Function definitions
def helper_functions():
    pass
# Main execution logic
if __name__ == "__main__":
    # Processing logic here
    result_dict = {"key": "value", "results": data}
# MANDATORY: Output results
output_results(result_dict)  # Must be final line
```

Listing 1: General Script Template Structure

A.2 MiniZinc Instance Feature Extraction Template guided by mzn-tuning-prompt.md

```python
# MANDATORY imports (exact format required)
from lmtune_helpers import input_data, output_results
import networkx as nx
```

```python
import numpy as np
def main():
    # Get instance data (no file I/O allowed)
    instance_data = input_data()
    # Initialize standardized results structure
    results = {
        "README": "~200 word methodology description",
        "characteristic_1": 0.0,  # Problem size metrics
        "characteristic_2": 0.0,  # Graph properties
        # ... (extract exactly 50 characteristics)
        "characteristic_50": 0.0  # Structural complexity
    }
    # Analyze constraint optimization instance
    # Extract solver-relevant characteristics:
    # - Problem size (variables, constraints)
    # - Graph properties (density, clustering, centrality)
    # - Data distribution (statistical properties)
    # - Structural complexity (symmetries, sparsity)
    # MANDATORY: Return standardized results
    output_results(results)
if __name__ == "__main__":
    main()
```

Listing 2: MiniZinc Instance Feature Extraction Template

A.3 Parameter Settings for different Algorithm Selectors

This section lists the details on parameter settings for different algorithm selectors. The following setting has been applied to both random forest training and LLAMMA with the random forest classification mode.

Parameter	Value	Discription
n_estimators	300	More trees for better performance on complex constraint patterns
max_depth	20	Deeper trees to capture complex constraint programming relationships
min_samples_split	5	
min_samples_leaf	2	
resampling_strategy	‘cv’	
resampling_strategy_arguments	{‘folds’: 5}	
max_features	‘sqrt’	Standard dimensionality reduction for tree diversity
class_weight	‘balanced’	Handle solver class imbalance in dataset
random_state	42	

Table 2: Random Forest Hyperparameters

Parameter	Value
time_left_for_this_task	User-specified (300-3600s)
per_run_time_limit	time_budget // 30 s
initial_configurations_via_metalearning	25
ensemble_size	50
resampling_strategy	‘cv’
resampling_strategy_arguments	{‘folds’: 5}
scoring_functions	[accuracy/ranking/Borda]
memory_limit	3072 MB
tmp_folder	Auto-generated
delete_tmp_folder_after_terminate	False
random_state	42

Table 3: AutoSklearn Standard Configuration

A.4 Experimental results on feature analysis

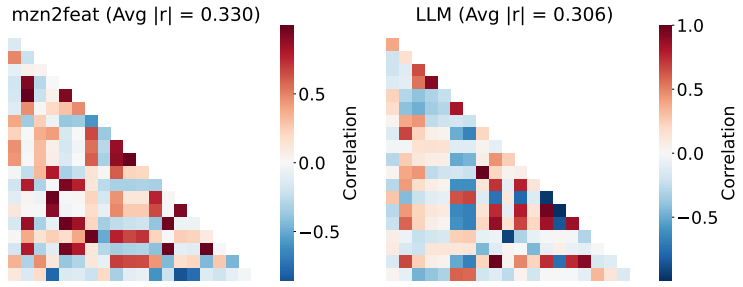


Figure 7: Feature correlation analysis for FLECC problem. LLM features show lower average correlation ($|r| = 0.306$) compared to *mzn2feat* ($|r| = 0.330$), indicating more diverse features.

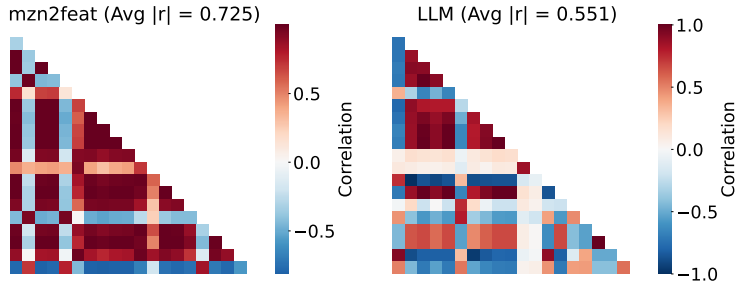


Figure 8: Feature correlation analysis for CS problem. LLM features show significantly lower average correlation ($|r| = 0.551$) compared to *mzn2feat* ($|r| = 0.725$), demonstrating 24%improvement in feature diversity.

A.5 Correlation Analysis on Problem-Generic Extractors and *mzn2feat*

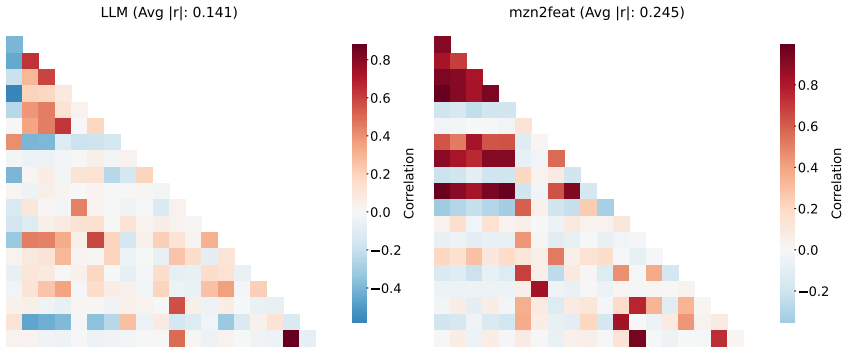


Figure 9: Feature correlation analysis for multiple problems. LLM features show significantly lower average correlation ($|r| = 0.141$) compared to *mzn2feat* ($|r| = 0.245$), demonstrating the improvement in feature diversity.

A.6 Top Features Distribution Analysis on Problem-generic Extractors and *mzn2feat*

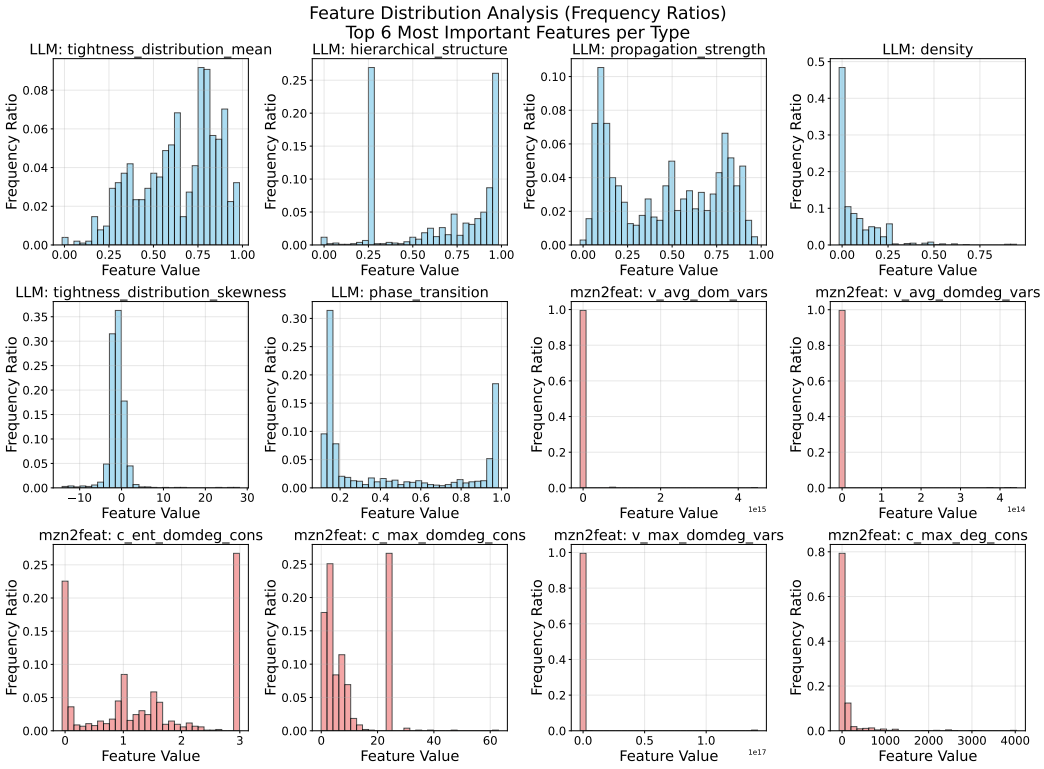


Figure 10: For LLM-based generic features and *mzn2feat*-based features, we compare the instance distribution on different single top features (Top-6 features). We can see that for LLM-based features, the instances are distributed more evenly according to the different values of the features, which potentially gives more useful information to the algorithm selectors for classification.

A.7 Experimental results of problem-generic feature extractor and *mzn2feat*-based algorithm selectors on training sets with Acc as loss functions.

Here we also show the results of *mzn2feat* and *LLM2feat*-based algorithm selectors' performance on training sets.

Loss function=Acc Training	VRP		CS		FLECC		Suite (200+Problems)		
	Acc	Rank	Acc	Rank	Acc	Rank	Acc	Rank	Borda
SB	78.7%	1.237	49.4%	1.601	80.6%	1.377	27.0%	2.667	0.694
mzn2feat+AF	78.7%	1.237	49.4%	1.601	80.6%	1.377	27.0%	2.667	0.694
mzn2feat+RF	91.7%	1.091	83.5%	1.231	88.3%	1.232	81.0%	1.186	1.491
mzn2feat+LLAMMA	92.0%	1.089	87.5%	1.248	91.5%	1.167	×	×	×
mzn2feat+AutoSK	87.2%	1.138	65.0%	1.723	85.0%	1.296	64.3%	1.322	1.402
LLM2feat+RF	97.1%	1.032	87.9%	1.164	95.9%	1.083	97.9%	1.016	1.653
LLM2feat+LLAMMA	97.8%	1.025	89.9%	1.164	98.3%	1.039	×	×	×
LLM2feat+AutoSK	87.8%	1.133	68.9%	1.618	95.1%	1.103	89.8%	1.078	1.608

Table 4: Experimental results of problem-specific and problem-generic feature extractors and *mzn2feat*-based algorithm selectors on training sets with Acc as loss functions.

A.8 Experimental results of problem-specific feature extractor and *mzn2feat*-based algorithm selectors on test sets with $Rank$ as loss functions.

Here we also show the results of *mzn2feat* and *LLM2feat*-based algorithm selectors' performance on testing sets with $Rank$ as the loss function.

Loss function=$Rank$ (Testing)	VRP		CS		FLECC	
	Acc	Rank	Acc	Rank	Acc	Rank
SB	79.5%	1.221	49.0%	1.616	78.2%	1.420
mzn2feat+RF	83.0%	1.184	58.5%	1.680	79.5%	1.391
mzn2feat+AutoSK	84.4%	1.166	62.1%	1.787	79.4%	1.394
LLM2feat+RF	85.7%	1.155	61.3%	1.681	83.5%	1.338
LLM2feat+AutoSK	85.4%	1.160	64.0%	1.738	84.6%	1.310

Table 5: Experimental results of problem-specific feature extractor and *mzn2feat*-based algorithm selectors on test sets with $Rank$ as loss functions.

A.9 Experimental results of problem-specific feature extractor and *mzn2feat*-based algorithm selectors on training sets with $Rank$ as loss functions.

Here we also show the results of *mzn2feat* and *LLM2feat*-based algorithm selectors' performance on training sets with $Rank$ as the loss function.

Loss function=$Rank$ Training	VRP		CS		FLECC	
	Acc	Rank	Acc	Rank	Acc	Rank
SB	78.7%	1.237	49.4%	1.601	80.6%	1.377
mzn2feat+RF	91.7%	1.091	83.5%	1.231	88.3%	1.232
mzn2feat+AutoSK	87.2%	1.138	65.0%	1.723	85.0%	1.296
LLM2feat+RF	97.1%	1.032	87.9%	1.164	95.9%	1.083
LLM2feat+AutoSK	87.8%	1.133	68.9%	1.618	95.1%	1.103

Table 6: Experimental results of problem-specific feature extractor and *mzn2feat*-based algorithm selectors on training sets with $Rank$ as loss functions.

A.10 LLM Model Selection and Sensitivity Analysis of *LLM2feat* Extractors

In general, we can use all LLM models as the backend of the agent. But we find our framework is not sensitive to the potential hallucination of LLM. To analyze the sensitivity, we select OpenAI o4-mini-2025-04-16 for the problem-specific framework, and generate three different feature extractors for each problem, and use the most classical tool chain, RF, and the most advanced tool chain, AutoSK, for algorithm selector training. As we can see in the following tables, in each table, we have three different LLM-based feature extractors (LLM-timestamp) for the same training setting. The accuracy and the average ranking of their algorithm selectors fluctuate in a quite small range (usually less than 2%).

Meanwhile, we also compare the Acc and $Rank$ obtained by different algorithm selectors based on *mzn2feat* and all three *LLM2feat* extractors. We can see that all *LLM2feat*-based algorithm selectors can always get better performance than mzn2feat-based algorithm selectors on different metrics (Acc and $Rank$).

Regarding the selection of LLM models in the problem-generic framework, we use Anthropic claude-sonnet-4-20250514. Here we have to deal with 227 problems extractor generation, which is conveniently tackled with the subagent functionality of Claude.

Table 7: *LLM2feat*+RF Performance for FLECC Problem

Feature Extractor	Loss Function	Train Acc	Test Acc	Train Rank	Test Rank
Single Best (gecode)		0.806	0.782	1.377	1.420
mzn2feat	accuracy	0.959	0.764	1.063	1.426
mzn2feat	ranking	0.883	0.795	1.232	1.391
LLM-20250908123730	accuracy	0.981	0.847	1.029	1.286
LLM-20250908123925	accuracy	0.996	0.818	1.008	1.353
LLM-20250908124149	accuracy	0.994	0.836	1.011	1.315
LLM-20250908123730	ranking	0.959	0.835	1.083	1.338
LLM-20250908123925	ranking	0.935	0.809	1.121	1.384
LLM-20250908124149	ranking	0.952	0.826	1.095	1.358

Table 8: *LLM2feat*+RF Performance for CS Problem

Feature Extractor	Loss Function	Train Acc	Test Acc	Train Rank	Test Rank
Single Best (cplex)		0.494	0.490	1.601	1.616
mzn2feat	accuracy	0.857	0.547	1.339	1.943
mzn2feat	ranking	0.835	0.585	1.231	1.680
LLM-20250908123608	accuracy	0.903	0.577	1.193	1.874
LLM-20250908123905	accuracy	0.913	0.578	1.181	1.861
LLM-20250908124041	accuracy	0.906	0.581	1.202	1.862
LLM-20250908123608	ranking	0.868	0.607	1.179	1.700
LLM-20250908123905	ranking	0.879	0.613	1.165	1.681
LLM-20250908124041	ranking	0.875	0.606	1.171	1.704

Table 9: *LLM2feat*+RF Performance for VRP Problem

Feature Extractor	Loss Function	Train Acc	Test Acc	Train Rank	Test Rank
Single Best (scip)		0.787	0.795	1.237	1.221
mzn2feat	accuracy	0.947	0.824	1.057	1.195
mzn2feat	ranking	0.916	0.830	1.091	1.184
LLM-20250908115627	accuracy	0.993	0.850	1.008	1.169
LLM-20250908121942	accuracy	0.994	0.848	1.007	1.170
LLM-20250908123205	accuracy	0.985	0.853	1.016	1.165
LLM-20250908115627	ranking	0.971	0.852	1.032	1.163
LLM-20250908121942	ranking	0.972	0.857	1.032	1.155
LLM-20250908123205	ranking	0.970	0.852	1.034	1.163

Table 10: *LLM2feat*+AutoSK Performance for FLECC Problem

Feature Extractor	Loss Function	Train Acc	Test Acc	Train Rank	Test Rank
Single Best (gecode)		0.806	0.782	1.377	1.420
mzn2feat	accuracy	0.850	0.792	1.295	1.396
mzn2feat	ranking	0.850	0.794	1.296	1.394
LLM-20250908124149	accuracy	0.952	0.844	1.101	1.308
LLM-20250908123730	accuracy	0.951	0.846	1.104	1.310
LLM-20250908123925	accuracy	0.908	0.831	1.187	1.342
LLM-20250908124149	ranking	0.951	0.845	1.103	1.308
LLM-20250908123730	ranking	0.951	0.846	1.104	1.310
LLM-20250908123925	ranking	0.911	0.836	1.183	1.322

Table 11: *LLM2feat*+AutoSK Performance for CS Problem

Feature Extractor	Loss Function	Train Acc	Test Acc	Train Rank	Test Rank
Single Best (cplex)		0.494	0.490	1.601	1.616
mzn2feat	accuracy	0.650	0.621	1.723	1.787
mzn2feat	ranking	0.650	0.621	1.723	1.787
LLM-20250908124041	accuracy	0.671	0.640	1.646	1.735
LLM-20250908123905	accuracy	0.679	0.634	1.667	1.781
LLM-20250908123608	accuracy	0.686	0.635	1.652	1.782
LLM-20250908124041	ranking	0.670	0.640	1.655	1.738
LLM-20250908123905	ranking	0.675	0.638	1.655	1.766
LLM-20250908123608	ranking	0.689	0.639	1.618	1.743

Table 12: *LLM2feat*+AutoSK Performance for VRP Problem

Feature Extractor	Loss Function	Train Acc	Test Acc	Train Rank	Test Rank
Single Best (scip)		0.787	0.795	1.237	1.221
mzn2feat	accuracy	0.872	0.844	1.138	1.166
mzn2feat	ranking	0.872	0.844	1.138	1.166
LLM-20250908121942	accuracy	0.851	0.852	1.161	1.162
LLM-20250908123205	accuracy	0.880	0.857	1.131	1.157
LLM-20250908115627	accuracy	0.872	0.854	1.138	1.160
LLM-20250908121942	ranking	0.851	0.852	1.161	1.162
LLM-20250908123205	ranking	0.878	0.854	1.133	1.160
LLM-20250908115627	ranking	0.859	0.850	1.152	1.162