Solving ML with ML: Evaluating the performance of the Monte Carlo Tree Search algorithm in the context of Program Synthesis

Bastiaan Filius[1]

Supervisor(s): Sebastijan Dumančić[1]**, Tilman Hinnerichs**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Bastiaan Filius
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumančić, Tilman Hinnerichs, David Tax

Abstract

Machine learning pipelines encompass various sequential steps involved in tasks such as data extraction, preprocessing, model training, and deployment. Manual construction of these pipelines demands expert knowledge and can be time-consuming. To address this challenge, program synthesis offers an automated approach to generate computer programs based on high-level specifications or examples. By leveraging program synthesis, the development of machine learning solutions can be expedited, leading to broader adaptability. A key element of program synthesis is the objective function, which guides the combinatorial search for a program that satisfies user-defined requirements. This study examines the performance of the Monte Carlo Tree Search (MCTS) algorithm in the realm of generating machine learning pipelines through program synthesis. The research investigates the method's efficacy, explores its findings in terms of accuracy, cost, variance, and execution time, and draws conclusions regarding the algorithm's potential and limitations. By analyzing the MCTS algorithm's performance, this research contributes to the advancement of automated machine learning pipeline generation and highlights the benefits and considerations associated with using program synthesis techniques.

1 Introduction

Machine learning has gained significant traction across various domains [28]. However, one of the challenges is the considerable time investment required to develop effective machine learning solutions [19]. A crucial component of machine learning pipelines is data preprocessing, which involves cleaning, transforming, and preparing raw data to improve model training and prediction accuracy. Determining the appropriate data preprocessing techniques, selecting the best machine learning algorithm, and tuning hyperparameters often involve extensive trial and error. In this context, an intriguing approach to address this challenge is the automatic construction of machine learning pipelines using program synthesis, which is explored in this paper.

Program synthesis is a subfield in the study of computer science where high-level specifications or user-provided examples are used to automatically generate computer programs. It consists of creating methods and algorithms to close the gap between the desired behavior of a program and how it is implemented. Program synthesis can greatly simplify software development, minimize programming errors, and allow people that are less experienced in programming to design solutions without having in-depth knowledge of programming by automating the program development process [13].

To synthesize a machine learning pipeline, a crucial step involves defining the necessary functionalities within a grammar. These functionalities act as the fundamental building blocks of the pipeline, encompassing various preprocessing operations and a classifier. Once the grammar is established, a search algorithm is employed to traverse it, facilitating the iterative construction of the pipeline. In this paper, we utilize the Monte Carlo Tree Search algorithm as the underlying search algorithm for pipeline synthesis. By leveraging this algorithm, we can effectively explore and generate optimal pipelines through an inductive process.

The Monte Carlo Tree Search (MCTS) is a heuristic search algorithm used in decision-making processes for problems with large search spaces. It builds a search tree by traversing and expanding nodes, representing states and actions. MCTS balances exploration and exploitation by selecting nodes based on statistical analysis. It performs random rollouts to estimate action outcomes and uses backpropagation to update node statistics. With each iteration, MCTS refines its estimates and converges towards high-quality actions. It has been successful in game playing, planning, optimization, and other domains, making it a popular choice for solving complex decision-making problems [7].

In the process of finding and constructing a valid pipeline, it becomes necessary to assess the quality of the generated pipelines. Previous studies, such as He et al. (2021) [14], have proposed techniques to accelerate evaluation and reduce the number of options to consider. Additionally, Nguyen et al. (2020) [20] introduced the concept of using a surrogate model for machine learning pipeline evaluation. In this research, we aim to integrate some of these solutions into the search algorithm's methodology. By incorporating these techniques, we can enhance the efficiency and effectiveness of pipeline evaluation, ultimately improving the overall performance of the algorithm.

To conduct the experiments and evaluate the performance of the MCTS algorithm, a diverse set of datasets is required. These datasets serve as input for the machine learning pipelines constructed by the algorithm. In this study, we collected datasets using OpenML, a popular online platform for machine learning and data mining. We collected datasets, considering diverse domains and complexity levels. This would allow us to evaluate the MCTS algorithm's performance across different problem spaces. We prioritized balanced datasets with well-defined target variables. By leveraging these diverse datasets, we simulated real-world scenarios and assessed the algorithm's ability to construct pipelines for a wide range of problems. The dataset collection ensured comprehensive evaluation of the algorithm's adaptability to various data characteristics.

The results will be compared to five other search algorithms: Breadth First Search [3], A* [11], Metropolis-Hastings [8], Very Large Neighbourhood [1], and a genetic algorithm search [18]. Subsequently, the research question is: How well does the Monte Carlo Tree Search algorithm perform in the context of program synthesis?

The paper is structured as follows: Section 2 presents the methodology used in this research, including the synthesis of machine learning pipelines using the Monte Carlo Tree Search algorithm. Section 3 describes the experimental setup, covering the datasets, hyperparameters, and performance metrics. In Section 4, we present and discuss the results obtained. The discussion of the findings, including a

comparison with other algorithms, is provided in Section 5. Ethical considerations related to the research are discussed in Section 6. Finally, Section 7 provides conclusions and outlines potential future work.

2 Methodology

The methodology employed in this study comprises five key steps. First, a dataset of input datasets is created, providing the foundation for pipeline creation. Next, a grammar is established to define the components and structure of the pipelines. The Monte Carlo Tree Search algorithm is then implemented to facilitate the search and construction process. To expedite pipeline generation and evaluation, strategies for speeding up these processes are employed. Finally, the generated pipelines are thoroughly evaluated to assess their performance and effectiveness.

2.1 Dataset

To facilitate the synthesis of pipelines, we curated a diverse dataset collection to serve as input, resulting in a useful benchmark. The datasets were obtained from the OpenML platform via its API, which is dedicated to open machine learning resources [23]. A typical machine learning dataset consists of a matrix-like structure, where rows represent individual samples or instances, and columns represent features or attributes. A fundamental characteristic of these datasets is the association of each sample with a target class or label, providing the basis for the machine learning algorithm's predictive or classification tasks [24].

The choice was made to include three categories of datasets. First, we included nine simple datasets that are smaller in size, containing 4 to 42 features, and 2 to 3 target classes. These datasets, as shown in Table 1, provide a solid foundation for evaluating the performance of synthesized pipelines. Second, we incorporated five complex datasets, ranging from 2 to 6 target classes, 128 to 5000 features, and between 2600 and 13910 entries. These datasets, visible in Table 2, introduce higher levels of complexity to thoroughly assess the capabilities of the synthesized pipelines. Last, we included five datasets that were used in related research [5][21][22], as presented in Table 3. The inclusion of these datasets from comparable program synthesis studies enables meaningful comparisons and further enhances the utility of the benchmark.

By carefully selecting and incorporating datasets from various sources, including those found in related research, our dataset collection has resulted in a comprehensive and representative benchmark. This benchmark provides a solid foundation for evaluating the performance and effectiveness of the synthesized pipelines in different real-world scenarios and challenges.

2.2 Grammar

A context-free grammar (CFG) is a formal grammar used to describe the syntax of a language. It consists of production rules that define how symbols can be combined. Terminal symbols represent actual words, while non-terminal symbols represent abstract syntactic categories. CFGs generate valid strings by applying production rules, starting from a designated start symbol [9]. They are used in areas like programming language design and natural language processing. CFGs provide a formal framework for language structure description and serve as a foundation for parsing algorithms and language processing techniques [10] [6].

Name	ID	Entries	Features	Target Classes
diabetes	37	768	8	2
qsar-biodeg	1494	1055	42	2
seeds	1499	210	7	3
wdbc	1510	569	30	2
iris	61	150	4	3
blood-transfusion	1464	748	4	2
monks-problems-2	334	601	6	2
ilpd	1480	583	5	2
tic-tac-toe	50	958	9	2

Table 1: Collection of simple datasets

Name	ID	Entries	Features	Target Classes
har	1478	10299	561	6
gisette	41026	7000	5000	2
madelon	1485	2600	501	2
musk	1116	6598	167	2
gas-drift	1476	13910	128	6

Table 2: Collection of complex datasets

We made use of the Herb.jl Julia framework on GitHub[1]. This framework contains packages for a collection of common utility functions and structures, functionality for declaring grammars, functionality for evaluating (candidate) programs, and search procedure implementations for the Herb Program Synthesis framework.

The grammar we constructed is based on the grammar found in Katz et al. (2020) [15]. One change we decided to make was the removal of NoOp(). This terminal specified that at that node no operation should be performed. However, every pipeline containing a NoOp() could be rewritten to an equivalent pipeline without the NoOp() operator. An example of this can be found in figure 1. Removing this operator allows for faster computation by reducing the search space. The grammar enables the creation of a directed acyclic graph (DAG) containing sequential and parallel processing operations. A DAG is a data structure composed of nodes connected by directed edges, where the edges have a defined direction and there are no cycles or loops within the graph. The parallel operations are concatenated allowing for ensembles of multiple classifiers.

The terminals of the grammar are operators from the Julia scikit-learn library [25]. The choice for scikit-learn was made as it is a powerful and one of the most commonly used libraries for machine learning [12]. To reduce the search space we made the decision to keep the number of operators limited. We included seven feature preprocessing operators, five feature selection operators, and five supervised classification operators. The final grammar can be seen in Figure 2.

[1]https://github.com/Herb-AI

Name	ID	Entries	Features	Target Classes
glass	41	214	9	6
car-evaluation	40664	1728	21	4
spambase	44	461	57	2
wine-quality-red	40691	1599	11	6
wine-quality-white	40498	4898	11	7

Table 3: Collection of datasets used in related research

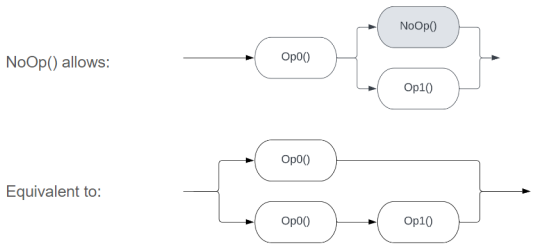

Figure 1: Equivalence of expressiveness after removing the NoOp() operator.

2.3 Search

The MCTS algorithm, implemented in Julia, consists of four main steps: selection, expansion, simulation, and backpropagation. Since MCTS is classically used in game theory related subjects some adaptions were necessary in its implementation for it to be applied in this research. The nodes in the tree represent pipeline configurations that are derived from the grammar. These derivations are done using a ContextFreeEnumerator from the Herb library which takes as an input: the grammar, the maximum depth, and the starting symbol. An overview of the algorithm in pseudocode can be found in Algorithm 1.

Algorithm 1: MCTS

Input: grammar, dataset, maxIterations, c
Output: bestPipeline, bestScore

```
bestScore ← 0;
bestPipeline ← nothing;
for i ← 1 to maxIterations do
    node ← rootNode;
    while !isempty(node.children) do
        node ← SelectChild(node, c);
    end
    if node.visits > 0 then
        node ← ExpandNode(node, grammar);
    end
    score ← Simulate(node.pipeline, dataset);
    if score > bestScore then
        bestScore ← score;
        bestPipeline ← node.pipeline;
    end
    Backpropagate(node, reward);
end
bestCost ← 1 - bestScore;
```

During the selection step, the algorithm employs the UCT

```
grammar = Herb.HerbGrammar.@cfgrammar begin
    START   = Pipeline([CLASSIF]) | Pipeline([PRE, CLASSIF])
    PRE     = PREPROC | FSELECT | ("seq", Pipeline([PRE, PRE])) |
    ("par", FeatureUnion([BRANCH, BRANCH]))
    BRANCH  = PRE | CLASSIF | ("seq", Pipeline([PRE, CLASSIF]))

    PREPROC =
        ("StandardScaler", StandardScaler()) |
        ("RobustScaler", RobustScaler()) |
        ("MinMaxScaler", MinMaxScaler()) |
        ("MaxAbsScaler", MaxAbsScaler()) |
        ("PCA", PCA()) |
        ("Binarizer", Binarizer()) |
        ("PolynomialFeatures", PolynomialFeatures())

    FSELECT =
        ("VarianceThreshold", VarianceThreshold()) |
        ("SelectKBest",  SelectKBest(k=4)) |
        ("SelectPercentile",  SelectPercentile()) |
        ("SelectFwe",  SelectFwe()) |
        ("Recursive Feature Elimination",  RFE(LinearSVC()))

    CLASSIF =
        ("DecisionTree", DecisionTreeClassifier()) |
        ("RandomForest", RandomForestClassifier()) |
        ("Gradient Boosting Classifier", GradientBoostingClassifier()) |
        ("LogisticRegression", LogisticRegression()) |
        ("KNearestNeighbors", KNeighborsClassifier())
end
```

Figure 2: Context-free grammar for generating machine learning pipelines.

(Upper Confidence Bound for Trees) selection policy [16] to traverse the tree from the root node to a leaf node. The UCT policy strikes a balance between exploration and exploitation by considering the estimated value and exploration potential of child nodes in the search tree. For the detailed pseudocode of the selection step and the UCT policy formula, refer to Algorithm 2. The UCT policy incorporates an exploration term that takes into account the visit count of a node, the total visit count of its parent, and an exploration constant c. This formula encourages the algorithm to explore less-visited nodes while favoring nodes with higher average rewards. Upon reaching a leaf node, the algorithm checks if it has been visited before. If so, the node is expanded by adding all possible pipeline derivations of one more depth as its child nodes, and one is randomly selected. An exploration constant analysis has been conducted to determine the optimal value for c, and the experimental setup and results are presented in Sections 3 and 4, respectively.

Next, in the simulation step, the algorithm classically performs a playout or rollout from the chosen node until a terminal state or predefined depth is reached. Since in the case of a machine learning pipeline a "win" or a "loss" cannot be achieved the configuration at that node is evaluated which produces an accuracy score.

Finally, in the backpropagation step, the results of the evaluated pipeline are backpropagated up the tree, updating the statistics of visited nodes. It increments the visited and score fields of the nodes by one and the accuracy score respectively.

By repeating these steps iteratively, MCTS explores the state space more effectively over time, focusing on more promising areas based on the collected statistics. The algorithm can be terminated after a certain number of iterations, and the best pipeline configuration is chosen based on the accumulated statistics.

Algorithm 2: SelectChild

Input: node, c
Output: bestChild

totalVisits ← sum(n.visits for n in node.children);
bestScore ← $-\infty$;
bestChild ← nothing;
for $child \in node.children$ **do**
  if *child.visits = 0* **then**
    score ← ∞;
  else
    explorationTerm ← c * sqrt(log(totalVisits) / child.visits);
    score ← child.wins / child.visits + explorationTerm;
  end
  if *score > bestScore* **then**
    bestScore ← score;
    bestChild ← child;
  end
end

2.4 Pipeline Generation and Evaluation

Executing pipelines during the search is an integral part of the simulation step within the MCTS algorithm. In each iteration of the search, a pipeline is generated and subsequently evaluated to estimate its performance.

To expedite the training process, techniques were employed to limit the pipeline training to the first *n* samples of a dataset. By using a subset of the data, the training time can be significantly reduced without compromising the overall evaluation accuracy. This approach, commonly known as "data subsampling," has been widely adopted in various AutoML solutions, including TPOT, Auto-WEKA, and Auto-sklearn.

These AutoML frameworks also utilize data subsampling techniques to improve computational efficiency during the pipeline search process. By training on a smaller subset of the data, these frameworks can explore a larger number of pipelines within a limited computational budget. This strategy is particularly advantageous when dealing with large datasets, where training on the entire dataset can be time-consuming and resource-intensive [14].

Additionally, to optimize the search process and avoid redundant computations, a dynamic programming technique was implemented. This technique involved storing pipelines that had been generated but not yet fully utilized, allowing them to be reused in subsequent iterations. By leveraging this storage mechanism, redundant computations associated with identical or similar pipeline structures were avoided, leading to improved efficiency and faster convergence.

The combination of training on a subset of data and the dynamic programming technique collaboratively enhance the efficiency of the pipeline generation and evaluation process within the MCTS algorithm. These optimizations not only reduce the computational cost but also expedite the convergence of the search towards high-quality pipelines.

2.5 Performance Evaluation

The evaluation of each generated pipeline was conducted using the "evaluate_pipeline" function. This function takes a pipeline and the train and validation sets as input. It employs the scikit-learn's "fit" function to train the model using the training set and generate predictions on the test set. The accuracy of these predictions was then computed and returned as the evaluation metric for the pipeline. In the case of invalid pipelines that could not be executed, a try-catch block was implemented within the evaluation function. This ensured that if an exception occurred during pipeline execution, an accuracy score of 0.0 was returned, preventing the influence of invalid pipelines on the overall accuracy calculation.

The cost of a pipeline was defined as 1 minus its accuracy, serving as a performance measure where lower values indicated higher accuracy. The "run_search" function played a crucial role in the performance evaluation. It accepted parameters such as dataset IDs, the number of runs, grammar details, enumeration and pipeline depths, subsampling size, and the number of MCTS iterations. By performing the MCTS algorithm for the specified number of runs and calculating the average accuracy over these runs, the function provided an overall performance metric for the algorithm.

3 Experimental Setup

In our experimental setup, we elaborate on the used data, the partitioning of the data, the chosen hyperparameters, and the metrics used in the results.

3.1 Training Data

For the hyperparameter analysis and evaluation, we focused on a subset of the benchmark datasets due to time and hardware constraints, as discussed in Section 5. Due to these limitations, only the diabetes, and spambase datasets were utilized for the hyperparameter analysis, and the seeds, wdbc, and har datasets were utilized for the evaluation. It is important to note that the seeds and wdbc datasets are relatively simple, while the har dataset presents a higher level of complexity. The choice to use different datasets for the hyperparameter analysis and for the evaluation was done to prevent overfitting, which occurs when the model becomes overly specialized to the characteristics of a specific dataset. If the same dataset is used for both hyperparameter tuning and performance evaluation, there is a risk of selecting hyperparameters that are optimized for that specific dataset but may not generalize well to new, unseen data. By using different datasets, you ensure that the hyperparameters are chosen based on their ability to perform well on a variety of data distributions. Although the dataset selection was limited, these datasets provided valuable insights into the behavior and effectiveness of the MCTS algorithm within the given constraints.

To ensure reproducibility, a seed value was used for shuffling the datasets. Julia has a built in feature to input a custom seed through its "Random" package. The seed used in this research is: 1234. This seed guaranteed that the same shuffling order was maintained across multiple runs, allowing for consistent evaluation and comparison of results. The datasets

were then shuffled and divided into three sets: train, test, and validation, with a ratio of 70:15:15, respectively. This partitioning strategy facilitated training the pipelines on the training set, evaluating their performance on the test set, and further validating the selected pipelines using the validation set.

3.2 Variable values

The exploration constant is the only hyperparameter of the MCTS algorithm and it determines the balance between exploration and exploitation. The original paper introducing UCB1 (Upper Confidence Bound), which is the policy used in UCT, determined that the minimum value of c should be $\sqrt{2}$ [2]. This value has been adopted theoretically as the standard value for c. However, the choice of the exploration constant depends on the specific problem and the desired trade-off between exploration and exploitation. It is typically determined through experimentation or domain knowledge. Different values of c can lead to different behaviors and performance characteristics of the algorithm. Therefore, a hyperparameter optimization based on two datasets has been performed for the exploration constant. The results are visible in Section 4.

Running the algorithm multiple times is essential for assessing its robustness and stability. It enables the observation of performance variability across different runs, providing a more reliable estimate of the algorithm's effectiveness. Averaging the results helps smooth out random fluctuations and outliers, yielding a more representative measure of performance. In this study, the algorithm was executed 10 times (*n_runs* = 10) to ensure sufficient assessment.

To expedite the training phase of the pipeline within each iteration, subsampling was employed. This involved limiting the number of samples (*n_samples*) used from the provided training set. Subsampling is commonly utilized to address computational and resource constraints in machine learning tasks. In this research, it was specifically implemented to alleviate the computational burden associated with training models on large datasets. The value of *n_samples* was set to 300.

Considering that the algorithms would be executed on different hardware setups, it was decided not to employ a time cutoff but rather limit the number of evaluated pipelines. This approach aimed to ensure a fair comparison between the algorithms. The maximum number of evaluated pipelines was set to 100 (*max_pipelines* = 100).

The chosen values for *n_runs*, *n_samples*, and *max_pipelines* were carefully determined to allow for the algorithm to be run within a realistic timeframe without compromising performance and the reliability of the results.

3.3 Metrics

To evaluate the performance of the MCTS algorithm and compare it with other algorithms, we selected specific metrics that reflect its effectiveness in generating high-quality pipelines. In our research, we considered three key performance metrics: accuracy, cost, and variance.

Accuracy: Accuracy is a widely used metric in machine learning that measures the proportion of correct predictions made by a model. In the context of our research, accuracy represents the effectiveness of the generated pipelines in correctly classifying or predicting outcomes. We calculate the accuracy by comparing the pipeline's predictions with the ground truth labels from the test set.

Cost: In addition to accuracy, we also considered the cost metric. The cost represents the inverse of accuracy, i.e., the misclassification rate or the proportion of incorrect predictions. By using the cost metric, we can evaluate the performance of the pipelines from a different perspective, focusing on the impact of incorrect predictions.

Variance: Variance is a measure of the variability or spread in the performance of the algorithm across multiple runs. It provides insights into the stability and consistency of the algorithm's results. By examining the variance, we can assess the robustness and reliability of the MCTS algorithm in generating pipelines across different experimental runs.

By considering accuracy, cost, and variance as performance metrics, we gain a comprehensive understanding of the algorithm's effectiveness, its ability to generate accurate pipelines, the cost of misclassifications, and the consistency of results. These metrics provide valuable insights into the performance characteristics of the MCTS algorithm and enable a thorough comparison with other algorithms under evaluation

4 Results

In this section, we present the results of our study, which includes the performance evaluation of the Monte Carlo Tree Search (MCTS) algorithm in generating machine learning pipelines. We conducted a hyperparameter analysis to fine-tune the algorithm's exploration constant and assessed its effectiveness across multiple datasets. Additionally, we compare the performance of MCTS with other search algorithms to gain insights into its relative performance and potential advantages.

4.1 Hyperparameter analysis

In order to determine the optimal value for the exploration constant (c), we conducted an analysis based on the average accuracy over 10 runs. The results, depicted in Figure 3, indicated that there was minimal variation for dataset 44. However, for dataset 37, the value of $c = \sqrt{2}$ yielded the highest average accuracy. As a result, we selected this value of c for further experiments and evaluations.

4.2 Performance MCTS

We conducted a comprehensive performance evaluation of the MCTS algorithm on multiple datasets to assess its effectiveness in generating pipelines. The evaluation was based on 10 independent runs on each of the three selected datasets, and the following performance metrics were analyzed: average accuracy, average cost, variance, and average execution time of which the results are visible in Table 4.

Across the 10 runs over the three datasets, the MCTS algorithm achieved an average accuracy of 0.9598, demonstrating its capability to generate pipelines that yield accurate predictions on the evaluation datasets. The high average accuracy underscores the effectiveness of the MCTS algorithm in

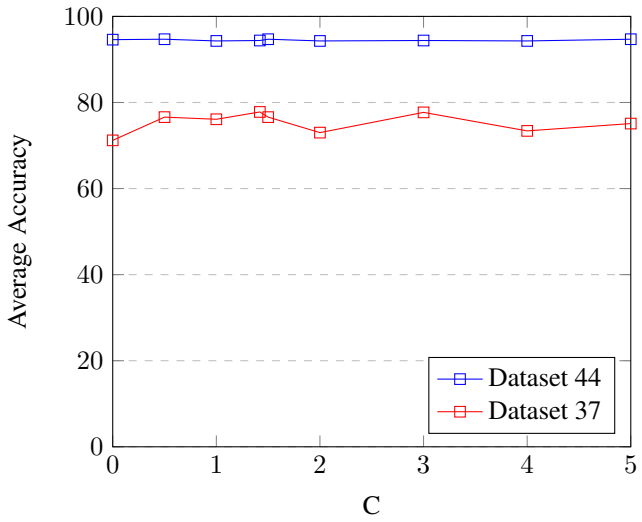

Figure 3: Hyperparameter optimization of the exploration constant.

Average over 10 runs	1499	1510	1478	Average
Accuracy	0.9281	0.9698	0.9814	0.9598
Cost	0.0719	0.0302	0.0186	0.040
Variance	0.00110	0.00034	0.00001	0.00051
Execution time	10.9 s	166.0 s	168.7 s	115.2 s

Table 4: Performance results of the MCTS algorithm.

exploring and selecting pipeline configurations that optimize predictive performance.

The cost metric, representing the misclassification rate or the proportion of incorrect predictions, was also calculated for the MCTS algorithm. The algorithm achieved an average cost of 0.0402, indicating its ability to minimize the impact of misclassifications and generate pipelines with improved predictive accuracy.

Assessing the variance of the MCTS algorithm's performance is crucial to understanding its stability and consistency across different runs. The variance analysis revealed a relatively low variance of 0.00051, suggesting that the algorithm consistently produces reliable results across the 10 runs. The low variance indicates that the algorithm is robust and less sensitive to random fluctuations or variations in the datasets.

The average execution time of our MCTS algorithm was measured to evaluate its computational efficiency. Over the 10 runs, the algorithm achieved an average execution time of 115.2 seconds, showcasing its capability to generate pipelines within a reasonable time frame. The efficient execution time is particularly beneficial for large-scale datasets, as it enables timely exploration of a wide range of pipeline configurations.

Overall, the MCTS algorithm demonstrates strong performance in terms of average accuracy, average cost, variance, and computational efficiency. The results highlight its potential as a valuable approach for automating machine learning pipeline generation, effectively balancing accuracy and computational resources.

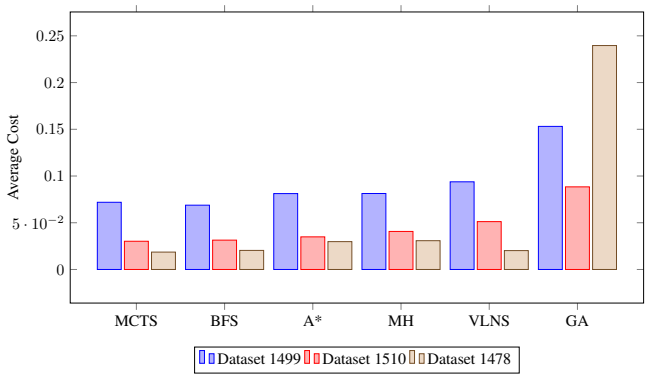

Figure 4: Comparison of Average Cost among Algorithms on Different Datasets

4.3 Comparison to other algorithms

To assess the effectiveness of the MCTS algorithm, its performance was compared with several algorithms from other research that used the same experimental setup, including Breadth-First Search (BFS), A* [17], Metropolis-Hastings (MH) [26], Variable Neighborhood Local Search (VLNS) [27], and Genetic Algorithm (GA) [4]. The comparison was conducted based on the same set of metrics used in the previous subsection.

When comparing the average accuracy across all algorithms, the MCTS algorithm got similar results or outperformed the other algorithms. In terms of the cost metric, our MCTS algorithm demonstrated favorable results. It achieved the same average cost as BFS, a difference of 0.0113 when compared with A*, MH, and VLNS, and a difference of 0.1202 when compared with GA, indicating its ability to generate pipelines with reduced misclassification rates and improved predictive performance. This highlights the effectiveness of the MCTS algorithm in minimizing errors and producing high-quality pipelines. The comparison is visible in Figure 4.

Analyzing the variance of the different algorithms revealed that the MCTS algorithm exhibited a lower variance for almost all datasets, with the only exception being BFS on dataset 1510, indicating greater stability and consistency in its performance across the 10 runs. The lower variance suggests that the MCTS algorithm consistently delivers reliable results, making it a robust choice for pipeline generation. The comparison is visible in Figure 5 (To enhance the clarity and readability of the figure, the decision was made to exclude the variance of the GA results, which was notably higher than other values by a factor of 10, in order to prevent it from overshadowing the results of other algorithms).

In order to ensure a fair and meaningful comparison, the decision was made not to compare the algorithms based on their execution time. This choice was motivated by the fact that the algorithms were executed on different hardware setups, making the results less reliable and not directly comparable.

Overall, the MCTS algorithm exhibits superior performance in terms of average accuracy, cost, and variance when compared to the BFS, A*, MH, VLNS, and GA algorithms.

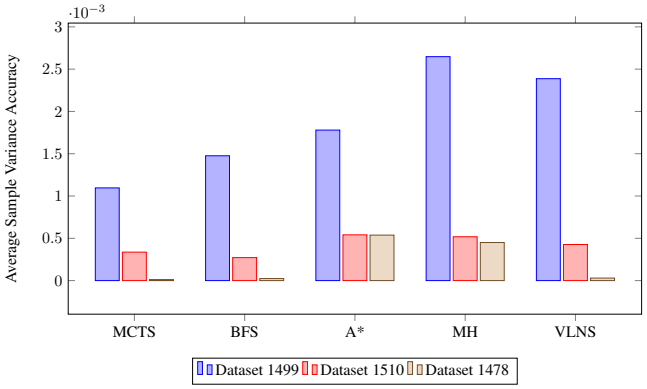

Figure 5: Comparison of Average Sample Variance Accuracy among Algorithms on Different Datasets

The results validate the effectiveness and efficiency of the MCTS algorithm as a promising approach for automating machine learning pipeline generation.

5 Discussion

The findings from our experimental analysis reveal notable performance advantages of the MCTS algorithm compared to alternative approaches, indicating its effectiveness within our research context. However, it is important to acknowledge the marginal improvements observed when comparing the MCTS algorithm with the BFS algorithm, which is recognized for its simplicity. This observation raises intriguing points for further discussion.

The relatively marginal gains of the MCTS algorithm over BFS could be attributed to the limitations of the datasets employed in our experiments. Due to constraints in time and resources, we were unable to assess the algorithm's performance on more diverse and complex datasets. Consequently, the algorithm's potential for significant advancements might have been underexplored. It is plausible that in complex problem domains, the MCTS algorithm could exhibit its superiority over BFS by utilizing its inherent capabilities for exploration and exploitation.

While our research focused on a restricted dataset selection, future investigations could broaden the evaluation to encompass a wider range of diverse and challenging datasets. This expanded assessment would enable a more comprehensive evaluation of the MCTS algorithm's performance and provide valuable insights into its effectiveness in addressing intricate problem spaces. Additionally, exploring further algorithmic enhancements and tailored adaptations specific to the problem domain could unlock the algorithm's full potential and lead to enhanced performance.

6 Responsible Research

This section addresses concerns related to the reproducibility and credibility of our research. Firstly, we discuss the origin and reliability of the datasets used in our study. Secondly, we highlight the measures taken to ensure reproducibility in our methodology. Lastly, we evaluate the credibility of the conclusions drawn from our research findings.

6.1 Data usage

Within this subsection, we aim to address concerns surrounding the datasets employed in our research. The selected datasets were sourced from OpenML, a well-established platform recognized for its provision of open datasets in the domains of machine learning and data mining. By leveraging datasets from OpenML, we prioritize transparency, accessibility, and reproducibility, benefiting from a centralized and dependable data source. Moreover, a subset of these datasets has garnered extensive utilization in analogous studies, attesting to their quality and suitability for our research. The integration of datasets sourced from OpenML, a renowned and established platform, bolsters the credibility and dependability of our research findings. Furthermore, we conducted rigorous assessments to ensure data integrity and alignment with the specific requirements of our study. By incorporating validated datasets previously employed in similar research, we increase the confidence in the usability and trustworthiness of our data.

6.2 Reproducibility

The Herb framework we used for our experiments is an open-source framework, which can easily be used for implementing and evaluating our algorithm. While the framework itself is readily accessible, it is important to note that the code used in our experiments will not be published in this paper. Instead, we provided pseudocode and describe the variables and parameters used in our algorithm. Although this approach may introduce some variation if others attempt to implement the algorithm on their own, we have taken care to ensure that all variables and their values are clearly mentioned in the paper. This transparency allows for a better understanding of the methodology and helps in the reproducibility of our experiments. However, due to the absence of complete code implementation, some variations may arise when reproducing our results. We encourage future researchers to refer to the Herb framework and the pseudocode provided in this paper as a starting point for their own implementations, adapting it to their specific needs and requirements.

6.3 Credibility

The credibility of our research findings is discussed in this subsection. While we draw conclusions based on our observations, it is essential to recognize the limitations and scope of our study. We acknowledge that the evaluation was conducted on a limited set of datasets, which may restrict the generalizability of our conclusions to broader problem domains. We emphasize the need for further research to expand the evaluation to include more diverse datasets and problem scenarios, thus enhancing the overall credibility and applicability of our findings.

7 Conclusions and Future Work

In conclusion, our experiments highlight the promising performance of the MCTS algorithm, although it showed marginal gains compared to the BFS algorithm The lack of substantial improvement can be attributed to the limited dataset complexity explored in this study. So to answer the

research question: How well does the Monte Carlo Tree Search algorithm perform in the context of program synthesis?, based on the results it shows potential but due to the limitations it is still uncertain whether it is applicable to more complex problems and datasets. Future research should delve into more challenging scenarios and consider incorporating domain-specific adaptations to further enhance the algorithm's effectiveness in real-world applications.

Moving forward, we recommend conducting experiments on more diverse and complex datasets to fully explore the capabilities of the MCTS algorithm. Additionally, further algorithmic refinements and modifications can be investigated to improve its performance, such as incorporating domain knowledge, refining the exploration-exploitation trade-off, or integrating other optimization techniques.

By addressing these aspects and expanding the scope of evaluation, future studies can provide a more comprehensive understanding of the MCTS algorithm's potential and its applicability in a wider range of problem domains. Such insights will contribute to the ongoing advancements in search algorithms and further propel the field of algorithmic optimization in the context of our research area.

How well does the Monte Carlo Tree Search algorithm perform in the context of program synthesis?

References

[1] Ravindra K Ahuja, Özlem Ergun, James B Orlin, and Abraham P Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3):75–102, 2002.

[2] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47:235–256, 2002.

[3] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[4] Mathieu Butenaerts, Tilman Hinnerichs, and Sebastijan Dumancic. Genetic algorithm-based program synthesizer for the construction of machine learning pipelines. *TU Delft preprint: available from repository.tudelft.nl*.

[5] José P Cambronero and Martin C Rinard. Al: autogenerating supervised learning programs. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–28, 2019.

[6] Eugene Charniak. Statistical parsing with a context-free grammar and word statistics. *AAAI/IAAI*, 2005(598-603):18, 1997.

[7] Guillaume Maurice Jean-Bernard Chaslot Chaslot. *Monte-carlo tree search*, volume 24. Maastricht University, 2010.

[8] Siddhartha Chib and Edward Greenberg. Understanding the metropolis-hastings algorithm. *The american statistician*, 49(4):327–335, 1995.

[9] Armin Cremers and Seymour Ginsburg. Context-free grammar forms. *Journal of Computer and System Sciences*, 11(1):86–117, 1975.

[10] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[11] Daniel Foead, Alifio Ghifari, Marchel Budi Kusuma, Novita Hanafiah, and Eric Gunawan. A systematic literature review of a* pathfinding. *Procedia Computer Science*, 179:507–514, 2021.

[12] Kaifeng Gao, Gang Mei, Francesco Piccialli, Salvatore Cuomo, Jingzhi Tu, and Zenan Huo. Julia language in machine learning: Algorithms, applications, and open issues. *Computer Science Review*, 37:100254, 2020.

[13] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[14] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.

[15] Michael Katz, Parikshit Ram, Shirin Sohrabi, and Octavian Udrea. Exploring context-free languages via planning: The case for automating machine learning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 403–411, 2020.

[16] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006: 17th European Conference on Machine Learning Berlin, Germany, September 18-22, 2006 Proceedings 17*, pages 282–293. Springer, 2006.

[17] Remi Lejeune, Tilman Hinnerichs, and Sebastijan Dumancic. Solving ml with ml: Effectiveness of a star search for synthesizing machine learning pipelines. *TU Delft preprint: available from repository.tudelft.nl*.

[18] Tom V Mathew. Genetic algorithm. *Report submitted at IIT Bombay*, page 53, 2012.

[19] M Arthur Munson. A study on the importance of and time spent on different modeling steps. *ACM SIGKDD Explorations Newsletter*, 13(2):65–71, 2012.

[20] Tien-Dung Nguyen, Tomasz Maszczyk, Katarzyna Musial, Marc-André Zöller, and Bogdan Gabrys. Avatar-machine learning pipeline evaluation using surrogate model. In *Advances in Intelligent Data Analysis XVIII: 18th International Symposium on Intelligent Data Analysis, IDA 2020, Konstanz, Germany, April 27–29, 2020, Proceedings 18*, pages 352–365. Springer, 2020.

[21] Randal S Olson, Nathan Bartley, Ryan J Urbanowicz, and Jason H Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the genetic and evolutionary computation conference 2016*, pages 485–492, 2016.

[22] Randal S Olson, William La Cava, Patryk Orzechowski, Ryan J Urbanowicz, and Jason H Moore. Pmlb: a large benchmark suite for machine learning evaluation and comparison. *BioData mining*, 10:1–13, 2017.

[23] OpenML, 2023. url=https://www.openml.org/.

[24] Amandalynne Paullada, Inioluwa Deborah Raji, Emily M Bender, Emily Denton, and Alex Hanna. Data and its (dis) contents: A survey of dataset development and use in machine learning research. *Patterns*, 2(11):100336, 2021.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[26] Denys Sheremet, Tilman Hinnerichs, and Sebastijan Dumancic. Solving ml with ml: Effectiveness of the metropolis-hastings algorithm for synthesizing machine learning pipelines. *TU Delft preprint: available from repository.tudelft.nl*, 2023.

[27] Auke Sonneveld, Tilman Hinnerichs, and Sebastijan Dumancic. Solving machine learning with machine learning: Exploiting very large-scale neighbourhood search for synthesizing machine learning pipelines. *TU Delft preprint: available from repository.tudelft.nl*.

[28] Aized Amin Soofi and Arshad Awan. Classification techniques in machine learning: applications and issues. *J. Basic Appl. Sci*, 13:459–465, 2017.