Solving ML with ML: Effectiveness of A Star Search for synthesizing machine learning pipelines

Rémi Lejeune[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: Rémi Lejeune
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumančić, Tilman Hinnerichs, David Tax

Abstract

This paper investigates the performance of the A* algorithm in the field of automated machine learning using program synthesis. We designed a context-free grammar to create machine learning pipelines and came up with a cost function for A*. Two different experiments were done, the first one to tune the parameters of our algorithm and the second one to compare the efficiency of A* with other search algorithms. The results indicate that for the selected datasets, A* did not have better performance, but rather had similar results with the other search algorithms. Nevertheless, more research in this field is needed to find concrete results.

1 Introduction

When trying to solve a problem with machine learning there are a lot of choices that need to be made. These include determining preprocessing steps that should be applied, in what order, or if they are necessary at all. Next, the choice of a machine learning model and the value of its hyperparameters is also needed, this increases the complexity of creating a machine learning pipeline. The amount of possible preprocessing functions, classifiers, and hyperparameters, which can be joined in various ways, opens up a complex and convoluted search space. Consequently, creating machine learning pipelines is a complex task that consumes lots of time, and requires advanced knowledge.

One approach is using autoML[6], the process of automating the tasks of applying machine learning to real-world problems. This is used in the TPOT[10] and AutoSklearn[3] paper. This field is still recent and has numerous potential improvements. Consequently, this paper uses autoML to create ML pipelines, in combination with Program synthesis.

Program synthesis[5] is the task of automatically finding a program that satisfies certain specifications. Most of the time its input is a list of input-output examples and it searches for a program that satisfies those requirements. However, in our case, we do not have input-output examples, our goal is to find the optimal pipeline. For this reason, the program outputted will be the best-performing pipeline found. This approach has already been developed in the papers: "Exploring Context-Free Languages via Planning: The Case for Automating Machine Learning" [7] and "Searching for Machine Learning Pipelines Using a Context-Free Grammar" [8]. But, one question still has not been answered: How do other search algorithms perform on autoML and how do they compare? Answering this question could improve the efficiency and accuracy of program synthesis to create pipelines.

Therefore, in this paper, we investigate whether A* improves the accuracy and efficiency of autoML using program synthesis. The following process will be applied to answer this question: select datasets to evaluate efficiency, design a context-free grammar that generates pipelines, choose a specific metric to assess pipelines, define a cost function for A*, and compare the efficiency of A* with other search algorithms.

This paper is divided into the following chapters: Chapter 2 is about methodology, it explains the process done to answer the question and explain the different choices. Chapter 3 describes the experimental setup and shows the results. Chapter 4 discusses the results and tries to draw conclusions from it. Chapter 5 is about responsible research. Chapter 6 concludes this paper and Chapter 7 discuss future works.

2 Methodology

This section discusses the methodology used to create our program. The first section talks about the datasets used to test and evaluate. The Second section introduces the grammar used to create pipelines. The third section explains which unit was to chosen to evaluate pipelines. Finally, the fourth section addresses the search function used and explain its underlying mechanism.

2.1 Datasets

To check the validity and effectiveness of pipelines, ensure the absence of errors, and reduce bugs, datasets are needed. They allow us to evaluate the performance of the synthesized pipelines and that our program is functioning correctly. A multitude of different datasets[1] is required to allow us to test the performance under different conditions, they are divided into three distinct groups1.

Simple datasets	Adv. datasets	Papers' datasets
Iris	gas-drift	glass
Seeds	musk	car-evaluation
Blood transfusion	madelon	WDBC
Monks-problem	gisette	wine-quality-red
Diabetes	har	spambase
Ilpd		wine-quality-white
Qsar-biodeg		
Tic-tac-toe		

Table 1: The 3 distinct groups of datasets used in this paper

The first category is simple datasets. These are chosen for their simplicity: a limited number of features (5-11), a small number of target classes (2 - 3), and a small size (150 - 768). These datasets are used to do basic tests, they are used to verify that the program runs without crashing our bugging. And more importantly, if the program manages to create near-optimal pipelines when working with simple datasets.

Next, once the program showed no issue with handling simple datasets, we designed a list of more complex datasets. They were selected to test the performance of the program under more complex scenarios. They possess a high number of features (26 - 1000), a broad range of classes (2 - 90), and a larger size (418 - 1.47m). The goal is to see how the program adapts to the datasets and performs when they become more complex.

[1] Source: OpenML website - https://www.openml.org

Lastly, we incorporated a list of datasets referenced in other autoML research papers[2][9]. This is done to provide us with a comparative platform. The goal is to compare our results with those found in other papers to see if our approach showed improvement and thus could be useful in the advancement of autoML.

2.2 Grammar

To be able to create machine learning pipelines, a Context-Free-Grammar was created. This grammar can create ML pipelines and is heavily inspired by the paper Exploring Context-Free Languages via Planning: The Case for Automating Machine Learning[7].

$$
\begin{aligned}
\langle \text{start} \rangle &\rightarrow \langle \text{dag} \rangle \gg \langle \text{est} \rangle \\
\langle \text{dag} \rangle &\rightarrow \text{NoOp}() \mid \langle \text{est} \rangle \mid \langle \text{tfm} \rangle \mid \langle \text{dag} \rangle \gg \langle \text{dag} \rangle \\
\langle \text{dag} \rangle &\rightarrow ((\langle \text{dag} \rangle)\{\,\&\,(\langle \text{dag} \rangle)\}^{+}) \gg \text{Concat}()
\end{aligned}
$$

Figure 1: Grammar from the paper "Exploring Context-Free Languages via Planning: The Case for Automating Machine Learning" [7]

The main difference between both grammars is the removal of *NoOp()* (No Operator), which is done because it is not needed and adds complexity to the grammar.

```
START =
   CLASSIF |
   sequence(PRE, CLASSIF)
PRE =
   PREPROC |
   FSELECT |
   sequence(PRE, PRE) |
   parallel(BRANCH, BRANCH)
BRANCH =
   PRE |
   CLASSIF |
   sequence(PRE, CLASSIF)
```

Figure 2: First iteration of our grammar

Figure 2 creates a grammar that can generate ML pipelines. This grammar enables pipelines to be in sequence but also to have different branches with multiple preprocessing steps and classifiers. Furthermore, the pipeline creation begins from the final step, ensuring that a classifier is always the last step.

The decision to maintain a simple grammar is motivated by several factors. Primarily, as the grammar grows exponentially, searching for long pipelines can be time-consuming since each of them needs to be evaluated. Furthermore, it facilitates easier implementation and debugging, an important consideration given our time limitations. Lastly, it enhances comprehensibility, making it easier to interpret and draw meaningful conclusions from it.

In our second iteration of the grammar, depicted in Figure 3, we solved several issues observed in the initial version.

Depth	Size
1	0
2	5
3	65
4	910
5	193515

Table 2: Relation between the depth and size of our first grammar

The revised grammar now generates a scikit-learn pipeline directly applicable for dataset evaluation, consequently enhancing usability. Moreover, a few modifications were made to the list of classifiers. Initially, XGBoost was included, but due to its incompatibility with the scikit-learn pipeline, we replaced it with the GradientBoostingClassifier from the same library. Furthermore, the SelectKBest function, which was previously set to select by default the top 10 features, was modified to select four instead. This adjustment was necessary since datasets with fewer than 10 features were incompatible with the original setting. By addressing these issues, we improved our grammar's utility, reducing crashes and bugs.

```
START =
   Pipeline([CLASSIF]) |
   Pipeline([PRE, CLASSIF])
PRE =
   PREPROC |
   FSELECT |
   ("seq", Pipeline([PRE, PRE])) |
   ("par", FeatureUnion([BRANCH, BRANCH]))
BRANCH =
   PRE |
   CLASSIF |
   ("seq", Pipeline([PRE, CLASSIF]))
```

Figure 3: Second iteration of our grammar

2.3 Evaluation

The metric employed to evaluate the effectiveness of pipelines is accuracy, it is chosen for its simplicity and interpretability. Accuracy allows us to have a high-level overview of the model's performance. Additionally, it can be easily implemented and is also a default measure in many machine learning frameworks. Moreover, it allows for an easy calculation of cost, defined as 1 - accuracy, a useful input for the A* algorithm. Nevertheless, we understand that accuracy can be misleading in scenarios with imbalanced datasets, for this reason, we ensured that all datasets used for experiments were balanced.

When running our different experiments, we divide each dataset into three distinct groups: training, validation, and testing. First, the training set is used to fit pipelines, then, the validation set assesses the performance of these trained pipelines and ranks them based on their effectiveness. This process prevents the program to overfit during the pipeline selection phase. Once, the best pipeline is found, it is evaluated a final time using the test set. Although this dataset

segmentation helps reduce overfitting and promotes generalization it doesn't guarantee immunity from potential biases related to sample distribution within the datasets.

In order to mitigate the potential issues related to sample distribution within datasets, we employ random shuffling of each dataset. To maintain consistency across all shuffling procedures, a specific seed has been selected, to guarantee uniformity in the shuffling process. Furthermore, we conduct an evaluation of each dataset ten times, thereby diluting the impact of any potential unfavorable shuffling. Hence, data splitting and shuffling cooperate to ensure the reliability and generalizability of our findings.

2.4 Search

Searching for the best pipeline is normally done by enumerating the grammar, evaluating each possible pipeline, and taking the pipeline that is performing the best. But doing that is time-consuming and requires a lot of calculating power. Could this be solved using another search function? This is what we aim to explore with the A* algorithm. To implement A*, a cost function and a heuristic function need to be defined.

For the cost function, the first idea was to use the length of the pipeline, however, this is not representative of the calculation power needed, as a short pipeline can be more demanding than a long one. Therefore the cost function chosen measures the time taken in milliseconds for a pipeline to evaluate a given dataset.

The heuristic function is intuitively defined by how well a pipeline performs. However, unless a pipeline achieves a perfect score (which is rare), it is hard to know when to stop the search. Since, if we evaluate all pipelines this leads us back to our original search algorithm. For this reason, only a subset of the grammar is evaluated.

Finally both the cost and heuristic need to be joined, given that the performance of the pipeline is more important than the time it takes, the following formula was designed:

$$(1 - acc) + 0.001 * (1 + log_{10}(T)) \tag{1}$$

Here, acc denotes accuracy, which when subtracted from 1, gives an inaccuracy value. The algorithm aims to minimize this, thereby pushing towards higher accuracy solutions. The cost function T (ms) is modified by a logarithm to ensure gradual growth, then scaled by a factor of 0.001. This formula ensures that the algorithm will favor pipelines that have both higher accuracy and lower evaluation time.

3 Experimental Setup and Results

This section explains the setup and results of both experiments. These experiments were also completed on other algorithms such as Breadth-First Search (BFS), Genetic Algorithm (GA)[1], Metropolis-Hastings (MH)[11], Monte Carlo (MC)[4], and Very Large Neighborhood Search (VLNS)[12], which allows for a comparison of their performance. It aims not only at making the experiments reproducible but also at showing the results and describing them.

3.1 Tuning the parameters

The goal of this experiment is to tune the parameters of the various search algorithms and observe their effect on performance. In the context of A*, the only parameter that was adjusted is the maximum depth of the pipelines.

The maximum depth of the pipelines was tuned among three different values: 3, 4, and 5. We chose to not include depth 2, as it solely returns classifiers and any depth exceeding 5 was omitted due to being too time-consuming. Unfortunately, due to the inability to access TUDelft supercomputer (DelftBlue), we limited the number of samples pipelines were trained on, the number of pipelines evaluated, and the pipelines could only be evaluated on simple datasets, diabetes[2], and spambase[3].

To reproduce this experiment, the file *experiment_setup.jl*[4] was run with the following parameters:

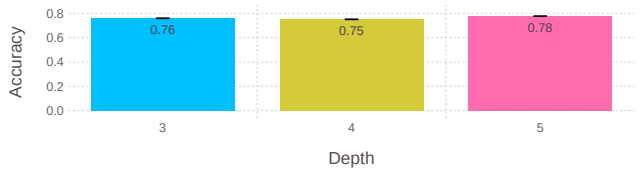
- train_on_n_samples = 300
- n_runs = 10
- max_pipelines = 100
- dataset_ids = [37, 44]
- search_alg_name = "astar"
- values = [3, 4, 5]

The outcome of this experiment is depicted in Figure 4. Focusing on the Diabetes dataset represented in subfigure 4a, it shows that depth 5 performed the best, closely followed by both depth 4 and 3, although the differences in accuracy between them are minimal. Observing the Spambase dataset from subfigure 4b, it is seen that all depths perform similarly. Thus, upon examining these two datasets, we can conclude that the depth value has an almost negligible influence on performance.

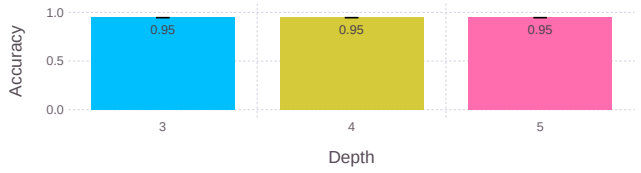
[2]https://www.openml.org/search?type=data&sort=runs&status=active&id=37

[3]https://www.openml.org/search?type=data&sort=runs&status=active&id=44

[4]https://github.com/MButenaerts/research_project/blob/main/experiment_setup.jl



(a) Diabetes dataset



(b) Spambase dataset

Figure 4: Results of tuning the maximum depth of the pipelines

3.2 Assessing the performance of A*

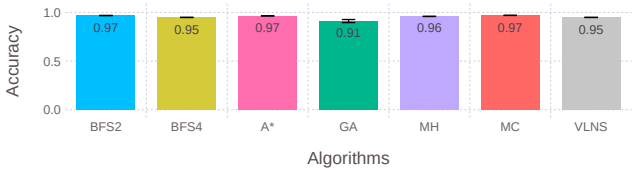
This experiment is created to compare the different search algorithms and determine whether the use of one, particularly A* in my case, had any advantages over the default search method (BFS).

For consistency, the various algorithms were executed on identical datasets. Owing to our inability to access Delft-Blue, this experiment was conducted solely on three diverse datasets: a simple one named 'seeds'[5], a complex one referred to as 'har'[6], and a third known as 'wdbc'[7] derived from a paper. Furthermore, on the outcome of our prior experiment, we set the maximum depth of the pipelines at 5, as this parameter proved to deliver the most favorable results.

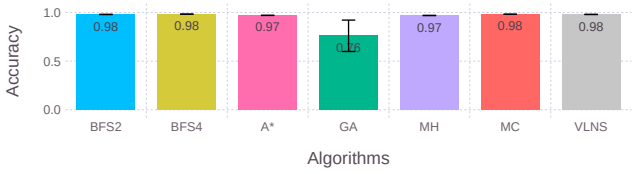
To reproduce this experiment, the file *experiment_setup.jl*[8] was run with the following parameters:

- train_on_n_samples = 300
- n_runs = 10
- max_pipelines = 100
- dataset_ids = [1499, 1510, 1478]
- search_alg_name = "astar"
- values = 5

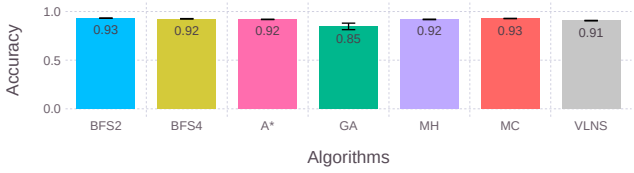
Figure 5 shows the accuracy of the different algorithms across the three different datasets. For subfigure 5a, Breadth-First-Search with depth 2 (BFS2) has the highest performance while Genetic Algorithm (GA) records the lowest, A* is closely trailing BFS2. Then, in the context of subfigure 5b, BFS4 outperforms all the other algorithms, GA is again performing the worst, and A* nearly matches the top performer. Additionally, subfigure 5c, displays a similar pattern to subfigure 5a with BFS2 leading, GA lagging, and A* closely following the leader. Overall, while BFS2 emerges as the superior choice on average, other algorithms like A* display performance close to the leading results.



(a) WDBC dataset



(b) HAR dataset



(c) Seeds dataset

Figure 5: Results of the different algorithms

Figure 6 represents the cost value of pipelines using Formula 1 $(1 - acc) + 0.001 * (1 + log_{10}(T))$ .These values are charted in the output order of the priority queue. Additionally, pipelines that failed to fit or evaluate the dataset were removed, which explains that the line does not extend to a hundred - the maximum pipeline value. Given our limitation of a hundred evaluations, not all pipelines have been evaluated. The three different graphs follow a similar pattern, where a majority of the evaluated pipelines exhibit low cost, with only the final 10-20% showing a high cost.

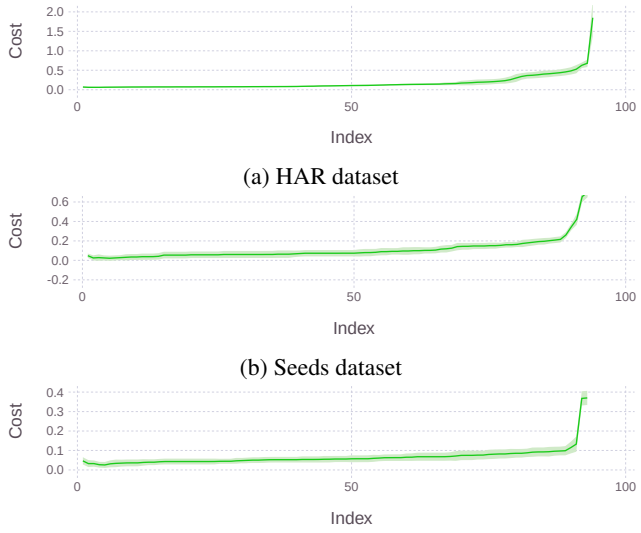
Figure 7 follows the same principles as Figure 6 but represents the time value of pipelines using the following formula: $0.001 * (1 + log_{10}(T))$, where T is measured in milliseconds. Again, the three different graphs reflect a comparable pattern, with most, if not all, pipelines displaying similar time values.

[5]https://www.openml.org/search?type=data&status=active&id=1499

[6]https://www.openml.org/search?type=data&status=active&id=1478

[7]https://www.openml.org/search?type=data&status=active&id=1510

[8]https://github.com/MButenaerts/research_project/blob/main/experiment_setup.jl

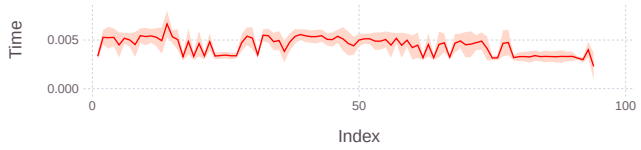


(a) HAR dataset

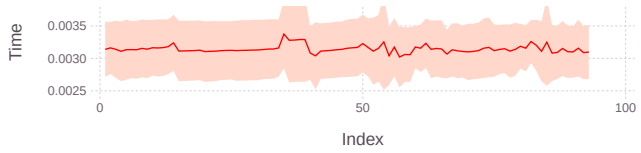
(b) Seeds dataset

(c) WDBC dataset

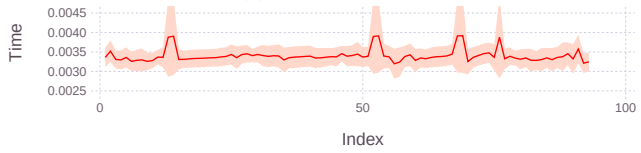
Figure 6: Cost value of pipelines, ordered by output order of the priority queue



(a) HAR dataset



(b) Seeds dataset



(c) WDBC dataset

Figure 7: Time value of pipelines, ordered by output order of the priority queue. The time is computed with the following formula: $.001 * (1 + log_{10}(T))$, where T is measured in milliseconds

4 Discussion

4.1 Tuning the parameters

From the results of the first experiment displayed in Figure 4, the conclusion can be drawn that the maximum depth of the pipelines has a negligible impact on performance. This suggests that preprocessing steps do not improve the performance of pipelines for the datasets used. This could be caused by the simplicity of these datasets.

4.2 Assessing the performance of A*

The results of the second experiment shown in Figure 5 can lead to the conclusion that A* doesn't outperform BFS, but has similar performance for the selected datasets. However, it's important to recognize that this observation cannot be generalized given that only three datasets underwent evaluation. The constant high performance of BFS2, which generates standalone classifiers without any preprocessing steps, suggests that there was no pressing requirement for preprocessing in these datasets.

Furthermore, both Figure 6 and Figure 7, indicate that both the cost and time values remain fairly consistent across the majority of the pipelines. This implies most pipelines evaluated by A* would likely yield positive results. However, this might be attributed to the fact that only a hundred pipelines were evaluated. A* started evaluating the neighboring pipelines of a well-performing pipeline and halted due to the imposed pipeline limit. As a result, it didn't have the opportunity to explore the breadth of the search space, concentrating mostly on the neighbors of an effective pipeline.

5 Responsible Research

5.1 Datasets

The datasets used in this paper are obtained via OpenML[9], which is a reputed website that provides datasets and other machine-learning tools. They were not manually verified, but are well-known and used by many other computer scientists.

5.2 Reproducibility

The code used is available on Github[10] and is open source. It will not be modified after the submission of this paper. Nonetheless, the code is using the Herb.jl[11] library, we cannot ensure that the library will stay compatible with our code in the sure.

5.3 Credibility

The results shown for the specific datasets can be considered credible. However, there is no proof that these results would extend to other datasets and the conclusion drawn from them might also not extend to other datasets, more experiment and testing is necessary to assess that.

6 Conclusion

The goal of this research paper was to investigate where A* improves the accuracy and efficiency of autoML using program synthesis. To achieve this, a series of steps were taken: datasets were selected to evaluate the efficiency, a context-free grammar was designed to generate machine learning pipelines, a metric was chosen to assess pipelines, and a cost formula was designed for A*. After this, two experiments were done, one to tune the parameters and a second to answer the research question by comparing the performance of A* with other search algorithms.

[9] https://openml.org

[10] https://github.com/M-Butenaerts/research_project

[11] https://github.com/Herb-AI/Herb.jl

From the last experiment, it can be concluded that A* does not necessarily improve accuracy and efficiency. However, it is important to note that these results are preliminary and further research is required.

7 Future Work

7.1 Datasets

This research was conducted with a limited number of datasets, doing more research with other datasets that have a higher complexity and require more preprocessing could be interesting. It would allow us to understand if the results found in this paper can be generalized more broadly.

7.2 A*

As discussed before, A* seems to get stuck in a specific branch and therefore does not have the opportunity to explore the breadth of the search space. Future research could design an adaptive version of A* to allow that, which could lead to an improvement in the efficiency of the algorithm.

7.3 Constraints

Two constraints were put in place to speed up the evaluation of the algorithms, train_on_n_samples, and max_pipelines. Future work could consider lifting these constraints, allowing pipelines to have a bigger training set, and evaluating more pipelines. A* performances should improve from these changes but it will also lead to an increase in the program's run time.

References

[1] 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*.

[2] José P. Cambronero and Martin C. Rinard. Al: Autogenerating supervised learning programs. *Proc. ACM Program. Lang.*, 3(OOPSLA):1–28, Oct 2019.

[3] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Autosklearn 2.0: The next generation. *arXiv preprint arXiv:2007.04074*, 24, 2020.

[4] Bastiaan Filius, Tilman Hinnerichs, and Sebastijan Dumancic. Solving ml with ml: Evaluating the performance of the monte carlo tree search algorithm in the context of program synthesis. *TU Delft preprint: available from repository.tudelft.nl*, 2023.

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

[6] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

[7] 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.

[8] Radu Marinescu, Akihiro Kishimoto, Parikshit Ram, Ambrish Rawat, Martin Wistuba, Paulito P Palmes, and Adi Botea. Searching for machine learning pipelines using a context-free grammar. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 8902–8911, 2021.

[9] 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.

[10] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*, pages 66–74. PMLR, 2016.

[11] 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.

[12] 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*.