Solving machine learning with machine learning: Exploiting Very Large-Scale Neighbourhood Search for synthesizing machine learning pipelines

Auke Sonneveld

Supervisor(s): Sebastijan Dumancic, Tilman Hinnerichs

[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 24, 2023

Name of the student: Auke Sonneveld
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumancic, Tilman Hinnerichs, David Tax

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Abstract

This paper presents a comparative study of multiple algorithms that can be used to automatically search for high-performing pipelines on machine learning problems. These algorithms, namely Very Large-Scale Neighbourhood search (VLSN), Breadth-first search, Metropolis-Hastings, Monte-Carlo tree search (MCTS), enumerative A* search, and Genetic Programming, are evaluated on three datasets. The performance of VLSN is consistently acceptable for this task, but the best performance is given by MCTS. Interestingly the results show that limiting the solution space to pipelines containing only a classifier operator does not significantly decrease performance. Three possible explanations for this are that the datasets used are too simple, the use of default hyperparameters makes preprocessing and feature selection operators useless, or the evaluation of pipelines on a limited training set makes the search procedure ineffective. This research contributes to the field of AutoML by shedding light on algorithm performance and providing insights for future improvements.

1 Introduction

Even though machine learning is now an integral part of modern technology, with countless applications ranging from recommendation systems [1] and protein folding [2] to large language models such as ChatGPT [3], applying machine learning to novel challenges still consumes a significant amount of time and requires expert knowledge. It is a challenging task to identify the necessary data pre-processing steps, determine the suitable machine learning algorithm, and select the optimal hyper-parameters. This complexity often creates a barrier for individuals without the necessary experience or expertise, limiting their ability to harness the potential of this powerful technology for solving their own unique problems.

Automated machine learning (AutoML) aims to alleviate these challenges by automating the process of model selection and optimization, as well as other tasks such as data preprocessing, meta-learning, and feature learning [4]. At the same time, program synthesis concerns itself with the task of automatically generating computer programs that satisfy some specified user intent. To increase the efficiency of program synthesis and interpretability of the generated program, a grammar can be provided [5], which dictates the set of programs that can be generated. When the grammar expresses machine learning pipelines the fields of AutoML and program synthesis intersect. In that case, an attempt is made to achieve AutoML, the automation of model selection and optimization, using program synthesis.

Despite significant progress in program synthesis and AutoML research, important questions remain unanswered. One of the critical issues that demands further investigation is the effectiveness of various search algorithms applied to a context-free grammar describing the domain of pipelines. This study aims to address this gap by investigating the performance of program synthesis using Very Large-Scale Neighborhood search (VLSN) when given such a grammar. The performance is compared to the performance of program synthesis using breadth-first search (BFS), Metropolis–Hastings (MH), Monte-Carlo tree search (MCTS), A* search (A*), and genetic programming (GP).

In this research, a collection of datasets and a context-free grammar are established, which are suitable for evaluating the performance of search algorithms in program synthesis. The evaluation setup used can be easily reproduced for further evaluation of the search algorithms, or adapted to evaluate the performance of additional search algorithms. Furthermore, the implementation of VLSN allows its utilization on any context-free grammar, provided the availability of a performance measure. Our findings demonstrate that VLSN achieves a slightly below-average performance on the simple datasets but a high performance on the complex dataset used. MCTS shows the highest performance across datasets, making it the most promising algorithm for this use case.

This research paper is structured as follows. Chapter 2 provides a comprehensive review of related work in the fields of AutoML and program synthesis. Chapter 3 presents the methodology employed in this study, covering aspects such as dataset construction, the context-free grammar utilized, the approach taken to evaluate pipeline performance, the implementation of VLSN, and the method of evaluating the program synthesizer. In Chapter 4, we present our experimental setup, specifying the runtime environment and the process of setting the hyperparameters of VLSN. The obtained results are detailed in Chapter 5. Chapter 6 offers an in-depth discussion of the findings and the limitations of the research, while Chapter 7 summarizes the main conclusions and discusses potential future work. Finally, in Chapter 8 responsible research considerations connected to the datasets, reproducibility, and credibility are addressed.

2 Related Work

Several notable works have contributed to the development of AutoML methodologies and tools, of which many are used or built upon in this research. The idea of AutoML, which focuses on addressing the challenges associated with automating the process of pipeline creation and model selection, has been around since the 1990s [6].

Hyperopt-Sklearn [7] introduced AutoML functionality to Python, leveraging the widely adopted Scikit-learn [8] package. Scikit-learn is a comprehensive machine-learning library that provides a range of algorithms and tools for data preprocessing, model building, and evaluation. It serves as the standard in AutoML research [4] and is utilized in this research.

The concept of model ensemble was introduced into AutoML by AUTO-SKLEARN [9] to enhance existing AutoML approaches. Model ensemble involves combining predictions from multiple machine learning models to improve overall performance and predictive accuracy.

TPOT [10] revolutionized the AutoML field by introducing Genetic Programming (GP) as a powerful method for automatically searching and evolving machine learning pipelines. It utilized a restricted set of Scikit-learn operators in its pipelines and achieved remarkable performance. In this research, GP serves as one of the algorithms VLSN is compared against and inspiration is drawn from TPOT's operator set in determining the grammar for this research.

Additionally, previous works proposed the use of context-free grammars to enable the creation of pipelines with arbitrary Directed Acyclic Graph (DAG) structures [11] [12]. The utilization of DAG-shaped pipelines enables greater flexibility and complexity in the construction of machine learning pipelines.

In the field of AutoML, various search algorithms have been utilized to automate the process of constructing machine learning pipelines. Examples include genetic programming employed by TPOT [10], Bayesian optimization utilized in Hyperopt-sklearn [7] and Auto-WEKA 2.0 [13], and Monte-Carlo Tree search explored in [14]. However, due to a scarcity of research specifically comparing the performance of multiple algorithms, it remains uncertain which algorithm performs best for the given task.

3 Methodology

In the following sections, an overview is provided of the components of the program synthesizer and argumentation is given for why they are designed in the way they are. Firstly, an overview is given of how the collection of datasets is compiled. Secondly, the context-free grammar used by the program synthesizer is explained. In the third section, we present the method of evaluating pipelines generated by the program synthesizer during the search. After that, the implementation details of the VLSN algorithm are given and lastly, we provide an overview of how the performance of the program synthesizer is evaluated.

3.1 Collection of Datasets

A total of nineteen classification datasets are collected, which are categorized into 8 simple datasets, 5 complex datasets, and 6 datasets that are used in similar research [15] [16] [17]. The OpenML [18] platform is used to obtain these datasets. OpenML provides a vast collection of datasets from various domains, making it a valuable resource for this research. The datasets, along with their ID on OpenML, and the number of entries, features, and target classes, can be found in Tables 1, 2, and 3.

Because of limited access to computational resources, only two datasets are used for hyperparameter optimization of the VLSN algorithm and only three datasets are used to evaluate the performance of the program synthesizer. The datasets used for hyperparameter optimization are "diabetes" and "qsar-biodeg", and the datasets used for evaluation of the algorithms are "seeds", "wdbc" and "har".

It is important to note that the ratio between any two target classes within the datasets used in this research is a 1:2 boundary, indicating a balanced distribution of classes. This does however not hold for all datasets in the collections.

3.2 Context-free Grammar Design

The context-free grammar should find a suitable trade-off between being expressive and efficient [5]. It should be expressive enough to encompass a broad range of machine learning

Name	ID	Entries	Features	Target Classes
diabetes	37	768	8	2
qsar-biodeg	1494	1055	42	2
seeds	1499	210	7	3
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

Name	ID	Entries	Features	Target Classes
wdbc	1510	569	30	2
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

pipelines while being restricted enough to facilitate efficient search.

The structure a pipeline can take and the operators a pipeline can use can be seen as the two defining dimensions of a context-free grammar expressing ML pipelines. In contemporary research, a diverse range of grammars is employed, exhibiting variations in both their structure and operators. Some grammars facilitate the construction of linear pipelines [19] [9], while others enable the creation of pipelines in the form of arbitrary Directed Acyclical Graphs (DAG) [12] [20]. Furthermore, there exists considerable diversity in terms of the operators utilized and the number of operators incorporated within these pipelines. Now an overview of the structure and operators implemented in the context-free grammar used in this research is given.

Structure

The context-free grammar facilitates DAG-shaped pipelines, with certain constraints. The derivation rules dictating the structure of the pipeline are given below. The | symbol is the divider between possible derivations.

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

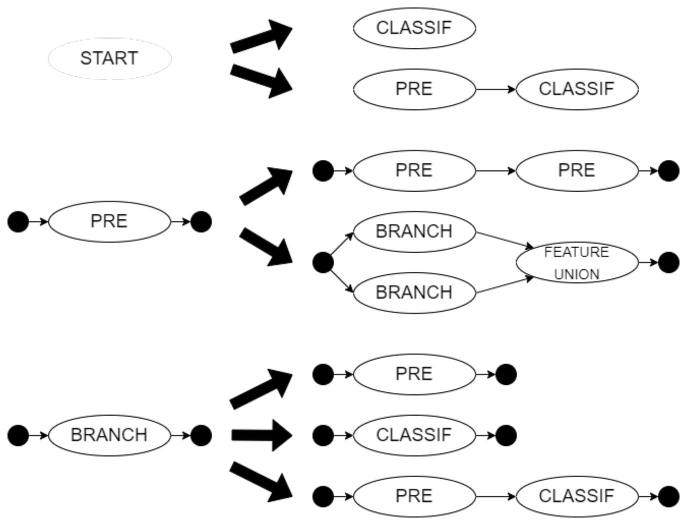


Figure 1: Node representation of grammar rules that are essential for the pipeline structure

Using these rules any arbitrary DAG shape can be created. The non-terminals *CLASSIF*, *PREPROC*, and *FSELECT* derive to a classifier operator, a feature preprocessing operator, and a feature selection operator respectively. A visual representation of these rules can be found in Figure 1. Note that *BRANCH* can be derived to *PRE*, allowing for an arbitrary amount of parallel branches.

A restriction that follows from the set of rules is that a classifier operator can only appear at the very end of the DAG or at the end of a branch. Therefore the output of a classifier is the final prediction or gets used in ensemble learning respectively. This is done to exclude pipelines in which the output of a classifier is the input of preprocessing or feature selection operator. The output of a classifier is a single classification, which is not a meaningful input to a preprocessing nor to a feature selection operator. Pipelines in which this occurs are thus nonsensical and should be avoided to increase efficiency without losing expressiveness.

Operators

A limited set of common operators from scikit-learn [8] is used to build the pipelines. This operator set is the same set as used in TPOT [10], with the only exception being that in this research the scikit-learn operator GradientBoostingClassifier is used instead of the separately imported XGBoost operator.

Compared to other operator sets, such as the operator set used in Auto-SKLearn [9] or RECIPE [20], the set of operators used by TPOT and thus also in this research is small. Nevertheless, [10] has shown that using this set of operators it is possible to obtain good performance on a wide variety of machine learning problems. Therefore it is sufficient for this research.

All operators are used with default parameters, except for SelectKBest for which k=4 is used. This is done because the default value k=10 surpasses the number of features in the ”diabetes” and ”seeds” datasets.

The seventeen operators used are divided into three categories based on their functionality. These categories are feature preprocessing operators, feature selection operators, and supervised classification operators. The operators used are:

- **Feature preprocessing operators:** StandardScaler, RobustScaler, MinMaxScaler, MaxAbsScaler, PCA, Binarizer, and PolynomialFeatures
- **Feature selection operators:** VarianceThreshold, SelectKBest, SelectPercentile, SelectFwe, and RecursiveFeatureElimination
- **Supervised classification operators:** DecisionTree, RandomForest, GradientBoostingClassifier, LogisticRegression, and NearestNeighborClassifier

Scikit-learn is used as algorithm source in four out of six AutoML systems analyzed in [4]. Using scikit-learn operators is thus in line with the standard in AutoML research.

3.3 Pipeline Evaluation Methodology

To make the search procedure more efficient only 300 entries of the training data are used to train models during the search. The evaluation of a pipeline involves training and validating the model. Training is computationally expensive, and evaluating numerous pipelines can quickly become time-consuming and unfeasible. Using limited training data is expected to not decrease the performance of the program synthesizer because for stochastic search algorithms like VLSN, knowing the direction of the search (based on relative pipeline performance) is more crucial than knowing the actual performance. In other words, using a limited training set decreases the performance of all pipelines, but as long as their relative performance remains the same the search procedure is not affected. The full training set is used when evaluating the final pipeline.

3.4 Very Large-Scale Neighborhood Search Implementation

Neighborhood search is a search algorithm that tries to find a good solution by repeatedly searching the neighborhood of the current solution and finding a better solution in that neighborhood [21]. In Very Large-Scale Neighborhood search the neighborhood is defined in such a way that makes it ’very large’.

This research uses Herb.jl [1], a Julia toolbox for program synthesis, for the construction of the neighborhood, and thus also adopts its definition of a neighborhood. In Herb.jl, to construct the neighborhood of a pipeline it is first represented as a tree structure. An example of a pipeline and its corresponding tree representation is given in Figure 2. In this tree representation, each branch node corresponds to an expression containing at least one non-terminal symbol defined in the grammar, while the leaf nodes represent terminal symbols. The neighborhood of a pipeline is created by randomly selecting a node from the tree and replacing it, along with all its descendant nodes, with various possible derivations of the non-terminal symbol located in its parent node. A leaf node can only have the non-terminal CLASSIF, PREPROC, or FSELECT above it and can thus be substituted only by other leaf nodes that can be derived from the same non-terminal.

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

Branch nodes can be substituted with much more complex sub-structures.

At the start of the search algorithm, a pipeline is randomly chosen from the pool of possible pipelines that can be constructed using the grammar. This pipeline is used to construct the neighborhood for the first iteration. In subsequent iterations, the best-performing pipeline in the neighborhood of the previous iteration is used to construct the neighborhood for the next iteration. In cases where the neighborhood does not contain a pipeline with superior performance compared to the current one but does contain a pipeline with equal performance, the current program is replaced with the equally performing pipeline. As a result, when the performance remains unchanged for multiple iterations, the exploration of the solution space continues. This allows for the potential discovery of a better solution.

The search process is terminated when 100 pipelines have been found and evaluated. More about this is explained in section 3.5.

The behavior of the search algorithm is influenced by the value of certain hyperparameters. These hyperparameters are:

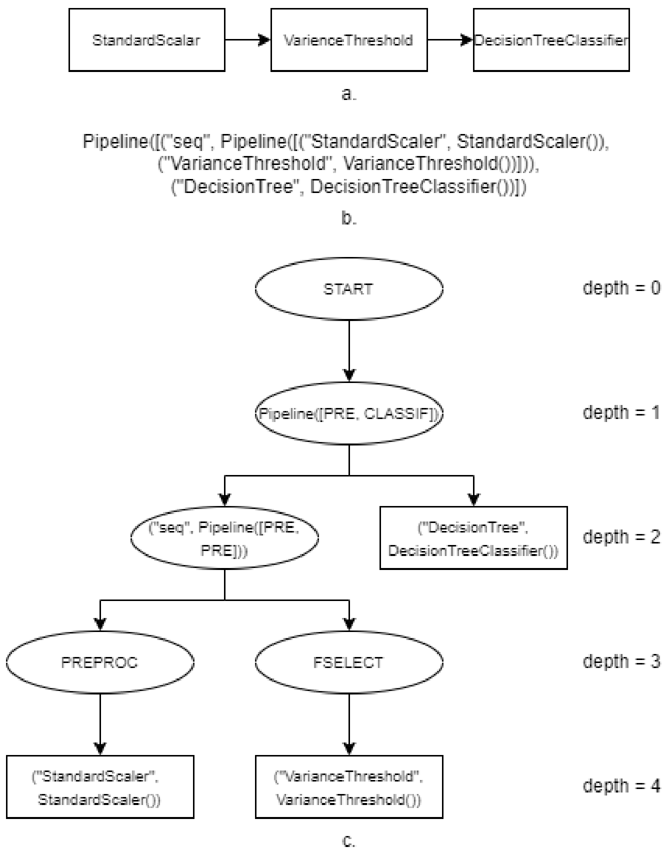
- *Neighbours per iteration* (npi): the number of neighbours to consider in each iteration.
- *Maximal allowed pipeline depth* (mapd): maximal depth of pipelines in the solution space.
- *Maximal allowed substitution depth* (masd): maximal depth of the substitute part of the pipeline.

In the context of this research, the depth of a pipeline refers to the depth of the tree representation of that pipeline. When constructing the neighborhood, the substitute part of the pipeline has to conform to the maximal allowed substitution depth and the new complete pipeline has to conform to the maximal allowed pipeline depth.

The best value for a hyperparameter is found by trying a variety of values for that hyperparameter whilst keeping the values of the other hyperparameters constant. To evaluate the performance of each value, the program synthesizer is run five times on each of the datasets ”diabetes” and ”qsar-biodeg”. Running it multiple times helps account for variance in individual results. After evaluating the performance of each value, the value that produces the best-performing pipeline is selected as the final value for that hyperparameter. This process is repeated for all hyperparameters until the best values are determined.

3.5 Performance Evaluation

The performance of the program synthesizer using VLSN is compared to the performance of the same program synthesizer using breadth-first search (BFS), Metropolis-Hastings (MH), Monte Carlo tree search (MCTS), A* search (A*), and genetic programming (GP). The performance measurements for MH, MCTS, A*, and GP are taken from [22], [23], [24], and [25] respectively. BFS is run once with mapd=2 and once without a limit on pipeline depth. When mapd=2 the solution space includes only five pipelines, which consist solely of a classification operator. This serves as a baseline comparison of how well the search algorithms perform against simply trying the five classifiers and picking the best. To ensure a fair


StandardScalar
VarienceThreshold
DecisionTreeClassifier
a.
b.
START
depth = 0
Pipeline([PRE, CLASSIF])
depth = 1
("seq", Pipeline([PRE, PRE]))
("DecisionTree", DecisionTreeClassifier())
depth = 2
PREPROC
FSELECT
depth = 3
("StandardScaler", StandardScaler())
("VarianceThreshold", VarianceThreshold())
depth = 4
c.

```
Pipeline([("seq", Pipeline([("StandardScaler", StandardScaler()),
("VarianceThreshold", VarianceThreshold())])),
("DecisionTree", DecisionTreeClassifier())])
```

Figure 2: Example pipeline representation. (a) Traditional schematic of the pipeline. (b) Grammar expression corresponding to the pipeline. (c) Tree representation of the pipeline's derivation.

and unbiased comparison, all algorithms are executed within an identical experimental environment, with all settings held constant except for the search algorithm itself.

To facilitate a fair comparison of the algorithms, it is necessary to ensure consistency even when they are executed on different machines. To achieve this, a standardized evaluation approach is adopted where each algorithm is terminated after evaluating 100 pipelines. This approach allows for a consistent benchmarking of their performance, regardless of variations in computational resources. Establishing a fixed evaluation threshold ensures that the comparison is conducted under the same conditions, enabling meaningful comparisons and insights into the algorithms' effectiveness.

The metric used to evaluate the performance of the algorithm is the accuracy of the final pipeline it produces on the test partition of the dataset. Accuracy is defined as the number of correct predictions divided by the total number of predictions made [26]. The choice of accuracy as the metric for evaluating prediction quality is appropriate in this study because the datasets are adequately balanced. The ratio between target classes is within 1:2 for all combinations of target classes, on all datasets. Furthermore, accuracy is chosen because of its simplicity and interpretability. Utilizing accuracy as the performance measure, we can directly compare the quality of pipelines produced by different search algorithms.

The datasets are split into a training set, validation set, and test set, using a ratio of 70:15:15. The training and validation sets are used to evaluate the performance of pipelines during the search procedure. Once the final pipeline is found, the pipeline is trained using the training set, and its performance is evaluated using the test set. The test set is thus never used before the final evaluation. By using different datasets for hyperparameter optimization than for the final evaluation of the algorithm, we further ensure the evaluation produces accurate results, as the test set is truly unseen.

To account for the inherent variability in the results of individual algorithm runs, we compute both the average and the standard deviation of the accuracy scores obtained from ten runs per dataset. This allows us to acquire a more reliable and robust comparison between the different algorithms.

4 Experimental Setup

This section outlines crucial components of the experimental setup to ensure reproducibility. First, an overview of the evaluation configuration is given, after which an in-depth explanation of how the hyperparameters of VLSN were determined can be found.

4.1 Evaluation Configuration

To compare the performance of VLSN to the other algorithms each algorithm is run ten times on three different datasets, where each run is allowed to evaluate 100 pipelines before being halted. VLSN is compared to BFS with a depth limit of 2, BFS without a depth limit, MH, MCTS, A*, and GP.

The datasets used during evaluation are ”seeds”, ”wdbc” and ”har”. Datasets ”seeds” and ”wdbc” are smaller in terms of entries, features, and target classes, while ”har” is more complex. All datasets are split into training, validation, and test set, using a ratio of 70:15:15. A seed is used for randomness to ensure the same splits are made for each algorithm.

Furthermore, to speed up the search, pipelines are only trained on 300 entries of the training data during the search. During the final evaluation, the complete training set is used. The hyperparameters for VLSN are set to npi=25, masd=4, and mapd=4. The implementations of MH, MCTS, A*, and GP, as well as the hyperparameters they used, are detailed in [22], [23], [24] and [25] respectively.

4.2 Parameters of VLSN

The hyperparameters *neighbours per iteration* (npi), *maximal allowed substitution depth* (masd) and *maximal allowed pipeline depth* (mapd) are optimized using the method explained in section 3.4 and their final values are npi=25, masd=4 and mapd=4. Figure 3 provides a visualization of the performance measures obtained during the optimization process. The average accuracy and standard deviation are computed from five runs per value per dataset.

During the optimization, each hyperparameter is individually optimized while keeping the others at their base values, which are npi=15, masd=3, and mapd=5. The search is terminated after 100 pipelines were evaluated. The datasets ”diabetes” and ”qsar-biodeg” are split into training sets, validation sets, and test sets, using a 70:15:15 ratio. The validation set is used to evaluate pipelines during the search, while the test set is solely utilized when the search has concluded to evaluate the found pipeline. A seed is used to ensure the splitting of the data was performed the same for all values of all hyperparameters.

The final value of npi is 25. On the ”diabetes” dataset, the average accuracy is 0.778 for npi=25, which is the highest of all the values tested. The next best average accuracy is achieved with npi=5, which gives an average accuracy of 0.764. However, it has a slightly higher standard deviation of 0.053 compared to 0.047 for npi=25. On the ”qsar-biodeg” dataset, npi=25 yields an average accuracy of 0.944, which is 0.006 lower than npi=5. Both npi=5 and npi=25 are good candidates with similar performance. When examining Figure 3a as a whole, except for the outlier on ”qsar-biodeg” with npi=15, it can be expected that any value between 5 and 25 for npi is suitable and achieves comparable average accuracy.

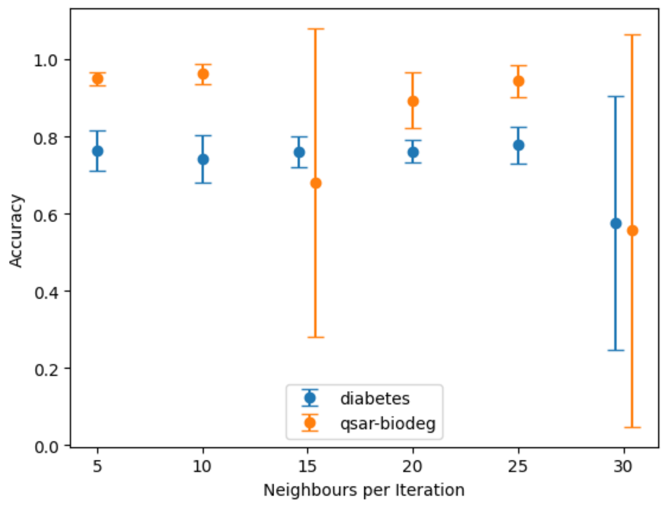
The final value of masd is 4. On the ”qsar-biodeg” dataset, masd=4 achieves the highest average accuracy of 0.9625. The second highest accuracy of 0.956 is obtained with masd=3, which performs poorly on the ”diabetes” dataset. For the ”diabetes” dataset, both masd=4 and masd=5 exhibit the same average accuracy of 0.748. However, masd=4 has a slightly higher standard deviation of 0.036 compared to 0.029 for masd=5. Based on the results, it is evident that masd=4 is the best option to choose. All results can be found in Figure 3b.

The value chosen for mapd is 4. It is clear when looking at Figure 3c that all values other than mapd=4 and mapd=5 result in lower average accuracy and higher standard deviation for at least one dataset. Therefore only mapd=4 and mapd=5 are suitable options. On ”diabetes” mapd=4 slightly outperforms mapd=5 and on qsar-biodeg, mapd=5 slightly outperforms mapd=4. To limit the complexity of the search space mapd=4 is chosen.

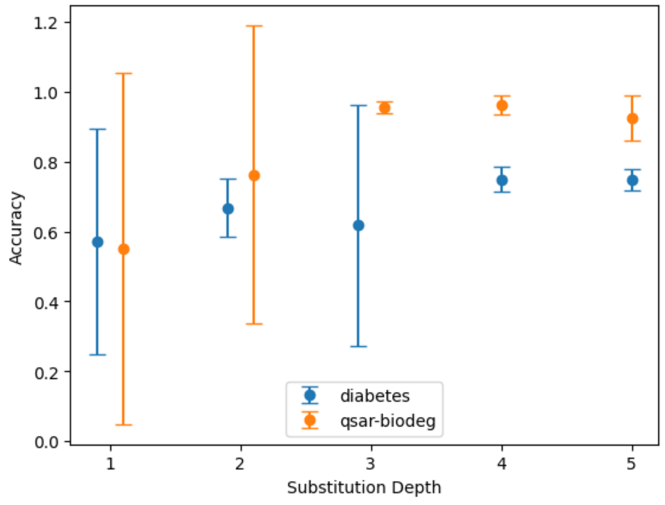
5 Results

The average accuracy and standard deviation achieved by the different algorithms on the ”seeds”, ”wdbc”, and ”har” datasets can be found in Table 4. A visual representation can be found in Figure 4, with Figure 4a showing the results on the ”seeds” database, Figure 4b showing the results on the ”wdbc” dataset, and Figure 4c showing the results on the ”har” dataset. The results of individual runs of VLSN, BFS(mapd=2), and BFS can be found in Appendix A

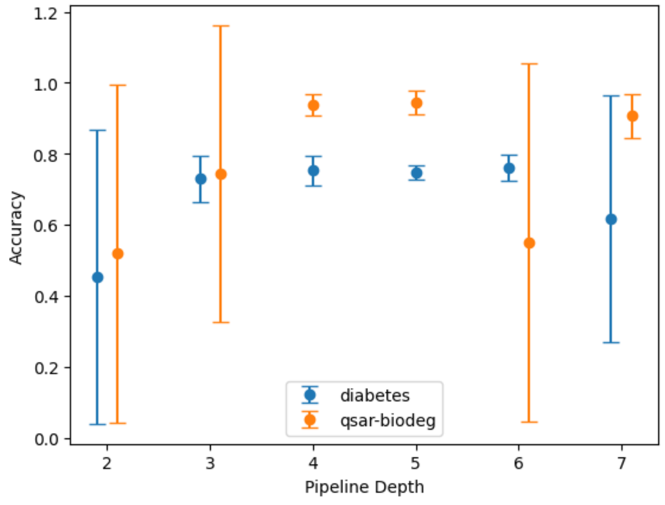
As can be seen in Table 4, BFS(mapd=2) slightly outperforms BFS on the ”wdbc” dataset but is slightly outperformed by it on the ”seeds” datasets. Interestingly, BFS(mapd=2) is not significantly outperformed by the other search algorithms on any dataset. On the ”seeds” dataset BFS(mapd=2) is outperformed by BFS and MCTS with a difference in average accuracy of 0.01, which is neglectable given the standard deviation. BFS(mapd=2) attains the same average accuracy of 0.92 as MH and A*. On the ”wdbc” dataset BFS(mapd=2) attains the highest average accuracy of 0.97, together with MCTS and A*. Similar results are seen on the ”har” dataset, where BFS(mapd=2), BFS, VLSN, and MCTS all achieve an



(a) Performance for different values of neighbours per iteration



(b) Performance for different values of maximally allowed substitution depth



(c) Performance for different values of maximally allowed pipeline depth

Figure 3: Average accuracy and standard deviation achieved on five runs for a range of values for VLSN's hyperparameters. Subfigures (a), (b), and (c) show the results for the different hyperparameters: (a) neighbours per iteration, (b) maximally allowed substitution depth, and (c) maximally allowed pipeline depth.

average accuracy of 0.98, while the other algorithms give a lower performance.

When specifically looking at the performance of VLSN compared to the other search algorithms we see that it is neither the highest nor the lowest-performing algorithm. MH and A* attain a slightly higher performance on the two simple datasets than VLSN, but perform worse on the "har" dataset. The difference in their average accuracy is within their standard deviation on all datasets. VLSN is slightly outperformed by MCTS on all datasets while it greatly outperforms GP on all datasets. Lastly, it achieves very similar, but sometimes slightly lower performance than BSF(mapd=2) and BFS.

When looking at the final pipelines outputted by the VLSN search an average pipeline depth of 3.6 is found on both the "seeds" and "wdbc" datasets, and an average pipeline depth of 3.3 is found on the "har" dataset. For the "seeds" and "wdbc" datasets this average is remarkably close to the maximal allowed pipeline depth of 4.

Finally, it can be seen that MCTS is among the highest-performing algorithms on all datasets. It consistently achieves equal or higher average accuracy than BFS(mapd=2), BFS, VLSN, MH, A*, and GP. These findings suggest that MCTS might be a superior algorithm for the use case of searching a context-free grammar for high-performance pipelines. It is however important to notice the difference in performance is often smaller than the standard deviation, weakening the claim.

	seeds		wdbc		ha	
Algorithm	A.A.	S.D.	A.A.	S.D.	A.A.	S.D.
BFS(mapd=2)	0.92	0.04	0.97	0.02	0.98	0.00
BFS	0.93	0.04	0.95	0.03	0.98	0.00
VLSN	0.91	0.05	0.95	0.02	0.98	0.01
MH	0.92	0.05	0.96	0.02	0.97	0.02
MCTS	0.93	0.03	0.97	0.02	0.98	0.00
A*	0.92	0.04	0.97	0.02	0.97	0.02
GP	0.85	0.17	0.91	0.12	0.76	0.38

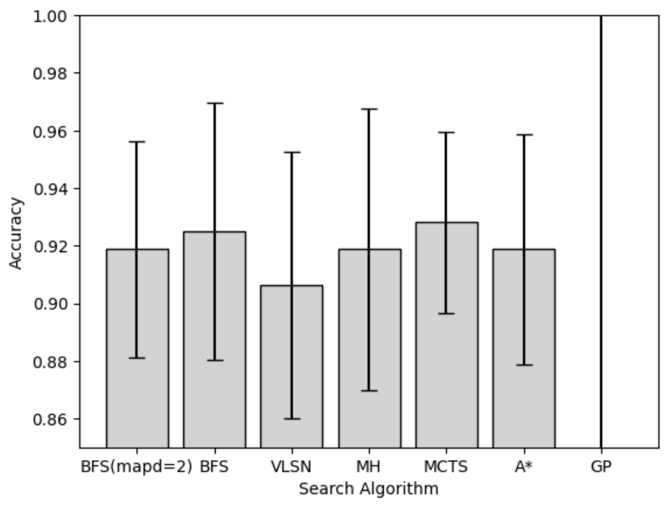
Table 4: Average Accuracy (A.A.) and Standard Deviation (S.D.) achieved by the different algorithms on the "seeds", "wdbc", and "har" datasets

6 Discussion

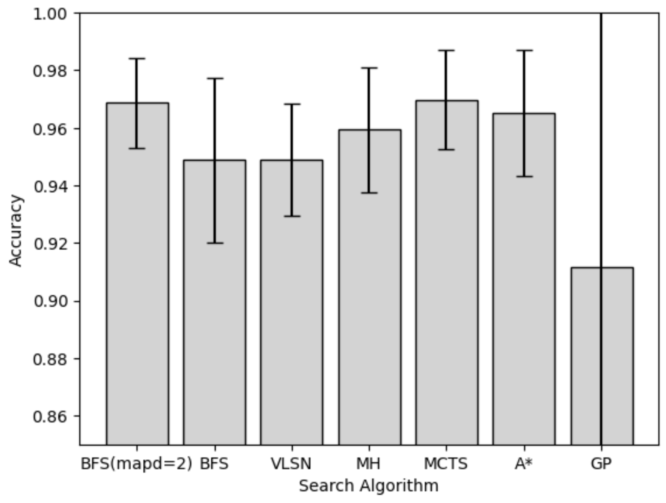
The discussion is separated into two parts. First, we discuss the result of the previous section, after which the limitations of this research are outlined.

6.1 Explanation of Results

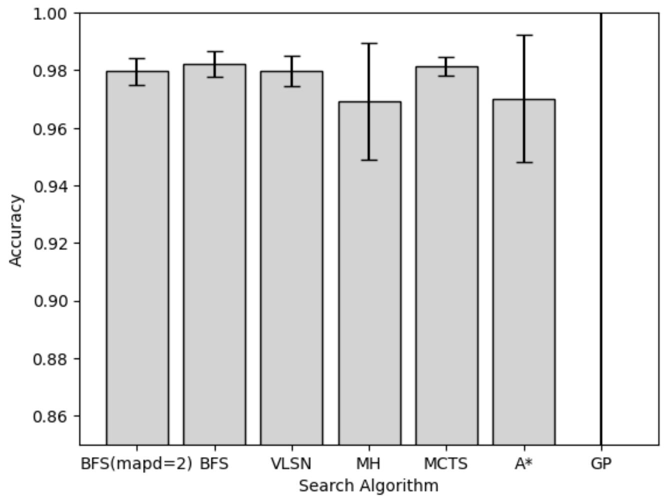
On first notice, it seems illogical that BFS(mapd=2) outperforms BFS on the "wdbc" dataset because the pipelines evaluated in BFS(mapd=2) are also evaluated in BFS. However, this can be explained by the fact that only 300 entries are used for training during the search. During its search, BFS evaluates 100 pipelines, and using this limited training set potentially decreases the accuracy of the evaluation. Of those 100 pipelines 5 consist of only a classifier and 95 are more



(a) Performance of the different algorithms on the seeds dataset



(b) Performance of the different algorithms on the wdbc dataset



(c) Performance of the different algorithms on the har dataset

Figure 4: Average accuracy and standard deviation achieved on ten runs by different search algorithms on three datasets. Subfigures (a), (b), and (c) show the results for the different datasets: (a) seeds dataset, (b) wdbc dataset, and (c) har dataset.

complex. Because of this ratio, the probability increases that a more complex pipeline is chosen by the algorithm based on its performance after being trained on 300 entries, even though it has a lower performance than a pipeline using only a classifier after being trained on the complete training set.

As shown in the results BFS(mapd=2) is not significantly outperformed by any of the search algorithms. By setting the mapd to 2, which limits the solution space, we are left with only 5 pipelines that exclusively comprise a classification operator. One possible explanation of these results is that data preprocessing and feature selection operators are not necessary for achieving high performance on these datasets. Generalizing this observation would lead to the conclusion that data preprocessing and feature selection operators are not necessary for datasets of comparable complexity to the ones used in this research. This is a radical conclusion as it eliminates the need for the very search algorithms we are evaluating and comparing.

Another explanation could be that search algorithms other than BFS (and BFS(mapd=2)) are unfairly affected by the use of a limited training set for the evaluation of a pipeline during the search procedure. The search path for all other algorithms is determined based on the evaluation of the pipelines it encounters during its search. During the search, only 300 samples are used for the evaluation of a pipeline and if that evaluation is therefore unreliable the algorithm might not find as good a pipeline. BFS would not be impacted by this in the same way because its search path is unaffected by the evaluation of pipelines it finds but instead predetermined.

A third and final explanation for the adequate performance of BFS(mapd=2) may lay in the utilization of default hyperparameters for the operators. For the feature selection and data preprocessing operators to function optimally suitable values have to be specified for their hyperparameters. Because the default values are used, the operators might lose their effectiveness which would explain why pipelines using solely a classifier operator are equally effective as pipelines using also feature selection or data preprocessing operators. This in turn would explain the similarity in performance of BFS(mapd=2) and the more advanced algorithms.

Furthermore, the depth of the final pipelines produced by VLSN is notable given the earlier observation that the most simple pipelines perform at least as well as more complex pipelines. The explanation for this occurrence finds its roots in the fact that there are many times more possible pipelines of depth *n* than of depth *n*-1. There are for instance 65 possible pipelines with depth 3, and there are 1505 possible pipelines with depth 4. Because the initial pipeline is randomly chosen from the set of all possible pipelines within the solution space, of which there are many more with the maximally allowed pipeline depth than any other pipeline depth, the initial pipeline is most likely to have the maximally allowed pipeline depth. For the same reason, when randomly choosing neighbours, these are more likely to be long substitutes with the maximal depth allowed than shorter pipelines. The combination of these two things causes short pipelines to rarely be evaluated by VLSN. Especially seeing the results indicate that short pipelines perform better, this imbalance might negatively impact the performance of VLSN.

When analyzing the performance of VLSN compared to the other algorithms, except for GP, on the simple datasets it performs slightly worse, but the difference is in all cases smaller than the standard deviation. When doing the same on the more complex "har" dataset we see it achieves the same, or slightly higher performance than the other algorithms, excluding GP. Again the difference is smaller than the standard deviation. Therefore based on this data, the performance of VLSN can be considered comparable to the performance of all other algorithms evaluated, except for GP.

As also stated in section 5 MCTS achieves high performance scores on all datasets. However, similar performance can be achieved on these datasets with the simple BFS(mapd=2). When more complex pipelines are required the performance of BFS(mapd=2) and BFS are expected to drop because of their inefficient search procedure, in which case MCTS emerges as the most promising algorithm. This is however based on the assumption that these results generalize to more complex datasets, which would need future research to confirm.

6.2 Limitations of the Research

The limitations of this research include the use of a limited number of datasets and runs, resulting in a significant standard deviation and potential bias in the observed performance. To improve the robustness and credibility of the results, a broader range of datasets and a larger number of runs should be considered. Additionally, the simplicity of the datasets used may limit the meaningful assessment of algorithm performance, warranting the inclusion of more diverse and complex datasets. This simplicity is highlighted by the adequate performance of BFS(mapd=2).

Furthermore, the exact implementation of VLSN can vary greatly, with one of the key aspects being the definition of the neighbourhood. The performance of different implementations might vary, and this variability applies to all algorithms compared in this paper, except for BFS and BFS(mapd=2). Therefore, it is important to note that any conclusions drawn can only be applied to the specific implementations used and cannot be generalized to the algorithms themselves.

Lastly, using the depth of the tree representation of a pipeline as a limiting factor creates a bias toward some derivations. Getting from PRE to two sequential operators takes one depth level less than getting to two parallel operators because the two BRANCH non-terminals have to be derived to PRE again before being derived to a terminal. This favors the sequential arrangement of operators compared to the parallel arrangement. In future research, this could be fixed by excluding non-terminals from the depth count.

7 Conclusions and Future Work

In this research, we compare several search algorithms, namely BFS, MH, MCTS, A*, GP, and VLSN, on the task of searching a context-free grammar to find high-performance pipelines for ML problems. The algorithms are evaluated on three datasets: "seeds", "wdbc", and "har".

The results reveal interesting findings regarding the performance of the algorithms. BFS with a depth limit of 2 (BFS(mapd=2)) achieves comparable performance to the more advanced algorithms on all datasets, suggesting that pipelines consisting solely of a classifier perform as well as more complex pipelines on the datasets used. This finding challenges the necessity of data preprocessing and feature selection operators for achieving high performance on datasets of comparable complexity.

Alternatively, the similar performance of BFS(mapd=2) compared to other search algorithms might indicate that the limited evaluation of pipelines during the search phase may impact the performance of more advanced algorithms. Furthermore, it could suggest feature preprocessing and data preparation operators are useless without hyperparameter optimization.

The VLSN algorithm, although not the highest-performing, demonstrates consistent, acceptable results. However, the preference for longer pipelines due to the search process's imbalance towards maximum depth pipelines may negatively affect its performance. MCTS achieves equal or higher performance than all compared algorithms on all datasets, indicating it might be a superior algorithm for searching context-free grammars for high-performance pipelines. However, due to the small difference in performance and the larger standard deviation observed this claim needs to be validated.

Nevertheless, it is important to acknowledge the limitations of this research. The use of a limited number of datasets and runs, as well as the simplicity of the datasets, may affect the generalizability of the results. The suspected variability in performance among different implementations of VLSN and the other algorithms further restricts the generalizability and limits the conclusions to the specific implementation used.

Further studies should explore a broader range of datasets, varying in complexity, and conduct a larger number of runs to enhance the robustness of the findings. Additionally, future research could investigate the reliability of pipeline evaluation on a limited number of samples, the impact of adding hyperparameter tuning for the operators, and the extension of the operator set to further improve pipeline performance.

In conclusion, this research contributes insights into the performance of various search algorithms for finding high-performance pipelines in context-free grammars. The findings challenge the necessity of data preprocessing and feature selection operators for datasets of comparable complexity and highlight the potential of BFS(mapd=2) and MCTS as efficient search algorithms. Future research can build upon these findings by considering more diverse datasets and refining the search algorithms' implementations to further enhance performance.

8 Responsible Research

Responsible research involves ensuring the reproducibility of the methods employed and questioning the credibility of the conclusions, thereby promoting transparency and scientific integrity. In this section, we first discuss the source and reliability of the datasets used, after which we critically reflect on the reproducibility of our research methods and outline the measures taken to facilitate the replication of our experiments. Lastly, the credibility of the drawn conclusions is questioned and the shortcomings are outlined.

8.1 Datasets

In this research, the datasets used were obtained from the OpenML platform, which serves as a reliable source for collecting diverse datasets from various domains. OpenML has established itself as a reputable resource in the field of machine learning, providing a vast collection of datasets that have been extensively used. The platform ensures transparency by providing detailed information about the datasets, including their origin and characteristics.

Furthermore, except for the ”seeds” dataset, all datasets used have tens or even hundreds of thousands of related runs on OpenML, increasing credibility. The fact that these datasets have been used so much increases the likelihood that they have undergone rigorous evaluation by other researchers. Lastly, no preprocessing was applied to the datasets, ensuring the integrity of the original data.

8.2 Reproducibility

Several measures have been taken to ensure reproducibility.

Firstly, all the code used in our research is available in a dedicated GitHub repository [2]. This repository includes the context-free grammar utilized in the program synthesis, as well as the implementation of the VLSN algorithm. The code is extensively documented, providing detailed explanations that aid in understanding the codebase and facilitate the reproduction of our research.

Furthermore, we have incorporated the Herb.jl framework in our research, which is also hosted on a GitHub repository. To ensure reproducibility, we have specified the exact branches and commits used in the setup, enabling researchers to precisely replicate the environment and configurations employed in our study.

Regarding computational resources, the choice was made to use the number of pipelines evaluated as a stopping criterion, rather than a time-based one. This allows reproducibility regardless of the computing power available.

By adopting these measures we aim to enhance the reproducibility of our research. We recognize that reproducibility serves as a cornerstone of scientific integrity and encourages future studies to build upon and validate our findings.

8.3 Credibility

While the conclusions drawn in this paper offer valuable insights into the research questions, their credibility is affected by the limitations in computational resources and the relatively small number of datasets and runs used. The credibility is further limited by the bias towards sequential pipelines inherent in the grammar. As such, these conclusions should be viewed as preliminary and call for future research efforts to validate and extend the findings. This can be done using more extensive resources and the collection of datasets presented in section 3.1.

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

References

[1] Shristi Shakya Khanal, P. W. C. Prasad, Abeer Alsadoon, and Angelika Maag. A systematic review: machine learning based recommendation systems for e-learning. *Education and Information Technologies*, 25(4):2635–2664, Jul 2020.

[2] Frank Noé, Gianni De Fabritiis, and Cecilia Clementi. Machine learning for protein folding and dynamics. *Current Opinion in Structural Biology*, 60:77–84, 2020. Folding and Binding Proteins.

[3] Yiheng Liu, Tianle Han, Siyuan Ma, Jiayue Zhang, Yuanyuan Yang, Jiaming Tian, Hao He, Antong Li, Mengshen He, Zhengliang Liu, Zihao Wu, Dajiang Zhu, Xiang Li, Ning Qiang, Dingang Shen, Tianming Liu, and Bao Ge. Summary of chatgpt/gpt-4 research and perspective towards the future of large language models, 2023.

[4] Thiloshon Nagarajah and Guhanathan Poravi. A review on automated machine learning (automl) systems. pages 1–6, 2019.

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

[6] Robin Hanson, John Stutz, and Peter Cheeseman. Bayesian classification theory. 05 1995.

[7] Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn. pages 32–37, 01 2014.

[8] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[10] Randal S. Olson and Jason H. Moore. *TPOT: A Tree-Based Pipeline Optimization Tool for Automating Machine Learning*, pages 151–160. Springer International Publishing, Cham, 2019.

[11] 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. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(10):8902–8911, May 2021.

[12] Michael Katz, Parikshit Ram, Shirin Sohrabi, and Octavian Udrea. Exploring context-free languages via planning: The case for automating machine learning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 30(1):403–411, Jun. 2020.

[13] Lars Kotthoff, Chris Thornton, Holger H. Hoos, Frank Hutter, and Kevin Leyton-Brown. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *Journal of Machine Learning Research*, 18(25):1–5, 2017.

[14] Herilalaina Rakotoarison, Marc Schoenauer, and Michèle Sebag. Automated machine learning with monte-carlo tree search (extended version). *CoRR*, abs/1906.00170, 2019.

[15] José P. Cambronero and Martin C. Rinard. Al: Autogenerating supervised learning programs. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[16] Randal S. Olson, Nathan Bartley, Ryan J. Urbanowicz, and Jason H. Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. *CoRR*, abs/1603.06212, 2016.

[17] 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):36, Dec 2017.

[18] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. Openml: Networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

[19] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. page 847–855, 2013.

[20] Alex De Sá, Walter Pinto, Luiz Otávio Oliveira, and Gisele Pappa. Recipe: A grammar-based framework for automatically evolving classification pipelines. pages 246–261, 03 2017.

[21] 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):75–102, 2002.

[22] 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.

[23] 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.

[24] 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*, 2023.

[25] 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.*

[26] Margherita Grandini, Enrico Bagli, and Giorgio Visani. Metrics for multi-class classification: an overview, 2020.

A Results of VLSN

In this appendix information about the runs of VLSN, BFS(mapd=2), and BFS is given. This includes the final pipeline that was found by each run, the test accuracy achieved by that pipeline, and the duration of the run. Tables 5, 6, and 7 show this information for the runs of VLSN on the ”seeds”, ”wdbc”, and ”har” datasets. Tables 8, 9, and 10 show this information for the runs of BFS(mapd=2) and Tables 11, 12, and 13 show this for the runs of BFS.

Run	Found Pipeline	Test Accuracy	Duration (seconds)
1	Pipeline([(”seq”, Pipeline([(”PCA”, PCA()), (”SelectKBest”, SelectKBest(k = 4))]))]), (”KNearestNeighbors”, KNeighborsClassifier())])	0.84375	5.17
2	Pipeline([(”seq”, Pipeline([(”SelectKBest”, SelectKBest(k = 4)), (”SelectKBest”, SelectKBest(k = 4))]))]), (”RandomForest”, RandomForestClassifier())])	0.9375	8.41
3	Pipeline([(”seq”, Pipeline([(”MinMaxScaler”, MinMaxScaler()), (”SelectKBest”, SelectKBest(k = 4))]))]), (”RandomForest”, RandomForestClassifier())])	0.90625	2.1
4	Pipeline([(”PolynomialFeatures”, PolynomialFeatures()), (”KNearestNeighbors”, KNeighborsClassifier())])	0.875	6.16
5	Pipeline([(”RobustScaler”, RobustScaler()), (”RandomForest”, RandomForestClassifier())])	0.9375	7.08
6	Pipeline([(”seq”, Pipeline([(”MinMaxScaler”, MinMaxScaler()), (”StandardScaler”, StandardScaler())]))]), (”KNearestNeighbors”, KNeighborsClassifier())])	0.90625	0.14
7	Pipeline([(”seq”, Pipeline([(”SelectKBest”, SelectKBest(k = 4)), (”PCA”, PCA())]))]), (”RandomForest”, RandomForestClassifier())])	0.84375	17.56
8	Pipeline([(”seq”, Pipeline([(”RobustScaler”, RobustScaler()), (”MinMaxScaler”, MinMaxScaler())]))]), (”KNearestNeighbors”, KNeighborsClassifier())])	0.875	0.08
9	Pipeline([(”Recursive Feature Elimination”, RFE(LinearSVC())), (”KNearestNeighbors”, KNeighborsClassifier())])	0.9375	9.65
10	Pipeline([(”Recursive Feature Elimination”, RFE(LinearSVC())), (”Gradient Boosting Classifier”, GradientBoostingClassifier())])	1.0	30.36

Table 5: Results of Very Large-Scale Neighbourhood search runs on the ”seeds” dataset

Run	Found Pipeline	Test Accuracy	Duration (seconds)
1	Pipeline([(”Recursive Feature Elimination”, RFE(LinearSVC())), (”RobustScaler”, RobustScaler())]), (”RandomForest”, RandomForestClassifier())])	0.953	20.59
2	Pipeline([(”Recursive Feature Elimination”, RFE(LinearSVC())), (”PCA”, PCA())]), (”RandomForest”, RandomForestClassifier())])	0.942	28.93
3	Pipeline([(”StandardScaler”, StandardScaler()), (”PCA”, PCA())]), (”Gradient Boosting Classifier”, GradientBoostingClassifier())])	0.953	47.96
4	Pipeline([(”MaxAbsScaler”, MaxAbsScaler()), (”PCA”, PCA())]), (”RandomForest”, RandomForestClassifier())])	0.930	2.81
5	Pipeline([(”SelectFwe”, SelectFwe()), (”Gradient Boosting Classifier”, GradientBoostingClassifier())])	0.942	26.06
6	Pipeline([(”MinMaxScaler”, MinMaxScaler()), (”PolynomialFeatures”, PolynomialFeatures())]), (”KNearestNeighbors”, KNeighborsClassifier())])	0.988	31.80
7	Pipeline([(”PolynomialFeatures”, PolynomialFeatures()), (”SelectPercentile”, SelectPercentile())]), (”DecisionTree”, DecisionTreeClassifier())])	0.942	61.06
8	Pipeline([(”PolynomialFeatures”, PolynomialFeatures()), (”MaxAbsScaler”, MaxAbsScaler())]), (”Gradient Boosting Classifier”, GradientBoostingClassifier())])	0.977	64.17
9	Pipeline([(”SelectFwe”, SelectFwe()), (”Gradient Boosting Classifier”, GradientBoostingClassifier())])	0.919	31.75
10	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.942	33.68

Table 6: Results of Very Large-Scale Neighborhood search runs on the ”wdbc” dataset

Run	Found Pipeline	Test Accuracy	Duration (seconds)
1	Pipeline([(”seq”, Pipeline([(”StandardScaler”, StandardScaler()), (”MinMaxScaler”, MinMaxScaler())]))]), (”LogisticRegression”, LogisticRegression())])	0.9799	270.46
2	Pipeline([(”PolynomialFeatures”, PolynomialFeatures()), (”RandomForest”, RandomForestClassifier())])	0.9735	405.2
3	Pipeline([(”SelectFwe”, SelectFwe()), (”LogisticRegression”, LogisticRegression())])	0.9806	438.18
4	Pipeline([(”SelectFwe”, SelectFwe()), (”LogisticRegression”, LogisticRegression())])	0.9806	265.96
5	Pipeline([(”StandardScaler”, StandardScaler()), (”RandomForest”, RandomForestClassifier())])	0.9806	150.61
6	Pipeline([(”StandardScaler”, StandardScaler()), (”LogisticRegression”, LogisticRegression())])	0.9793	149.15
7	Pipeline([(”PolynomialFeatures”, PolynomialFeatures()), (”LogisticRegression”, LogisticRegression())])	0.9890	468.55
8	Pipeline([(”seq”, Pipeline([(”VarianceThreshold”, VarianceThreshold()), (”VarianceThreshold”, VarianceThreshold())]))]), (”RandomForest”, RandomForestClassifier())])	0.9767	171.13
9	Pipeline([(”seq”, Pipeline([(”SelectFwe”, SelectFwe()), (”PolynomialFeatures”, PolynomialFeatures())]))]), (”RandomForest”, RandomForestClassifier())])	0.9709	452.04
10	Pipeline([(”PCA”, PCA()), (”LogisticRegression”, LogisticRegression())])	0.9871	667.0

Table 7: Results of Very Large-Scale Neighbourhood search runs on the ”har” dataset

Run	Found Pipeline	Test Accuracy	Duration (seconds)
1	Pipeline([(”DecisionTree”, DecisionTreeClassifier())])	0.90625	1.82
2	Pipeline([(”Gradient Boosting Classifier”, GradientBoostingClassifier())])	1.0	0.31
3	Pipeline([(”DecisionTree”, DecisionTreeClassifier())])	0.9375	0.3
4	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9375	0.3
5	Pipeline([(”Gradient Boosting Classifier”, GradientBoostingClassifier())])	0.96875	0.32
6	Pipeline([(”LogisticRegression”, LogisticRegression())])	0.875	0.34
7	Pipeline([(”Gradient Boosting Classifier”, GradientBoostingClassifier())])	0.90625	0.3
8	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.96875	0.3
9	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.90625	0.3
10	Pipeline([(”Gradient Boosting Classifier”, GradientBoostingClassifier())])	0.90625	0.3

Table 8: Results of BFS(mapd=2) runs on the ”seeds” dataset

Run	Found Pipeline	Test Accuracy	Duration (seconds)
1	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9534883720930233	0.58
2	Pipeline([(”LogisticRegression”, LogisticRegression())])	0.9767441860465116	0.46
3	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9767441860465116	0.45
4	Pipeline([(”RandomForest”, RandomForestClassifier())])	1.0	0.47
5	Pipeline([(”Gradient Boosting Classifier”, GradientBoostingClassifier())])	0.9418604651162791	0.47
6	Pipeline([(”Gradient Boosting Classifier”, GradientBoostingClassifier())])	0.9651162790697675	0.46
7	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9534883720930233	0.47
8	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9767441860465116	0.48
9	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9651162790697675	0.48
10	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9767441860465116	0.48

Table 9: Results of BFS(mapd=2) runs on the ”wdbc” dataset

Run	Found Pipeline	Test Accuracy	Duration (seconds)
1	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9695989650711514	29.46
2	Pipeline([(”LogisticRegression”, LogisticRegression())])	0.981888745148771	28.84
3	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9741267787839586	28.66
4	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9786545924967659	28.67
5	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9786545924967659	28.67
6	Pipeline([(”LogisticRegression”, LogisticRegression())])	0.98124191461837	28.58
7	Pipeline([(”LogisticRegression”, LogisticRegression())])	0.9851228978007762	28.63
8	Pipeline([(”LogisticRegression”, LogisticRegression())])	0.9805950840879689	28.89
9	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9799482535575679	28.93
10	Pipeline([(”LogisticRegression”, LogisticRegression())])	0.9864165588615783	28.61

Table 10: Results of BFS(mapd=2) runs on the ”har” dataset

Run	Found Pipeline	Test Accuracy	Duration (seconds)
1	Pipeline([(”LogisticRegression”, LogisticRegression())])	0.9375	10.03
2	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.875	7.91
3	Pipeline([(”seq”, Pipeline([(”SelectKBest”, SelectKBest(k = 4)), (”PCA”, PCA())])), (”LogisticRegression”, LogisticRegression())])	0.90625	7.87
4	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.96875	6.64
5	Pipeline([(”SelectKBest”, SelectKBest(k = 4)), (”LogisticRegression”, LogisticRegression())])	0.90625	6.75
6	Pipeline([(”seq”, Pipeline([(”VarianceThreshold”, VarianceThreshold()), (”PolynomialFeatures”, PolynomialFeatures())])), (”RandomForest”, RandomForestClassifier())])	0.875	1.36
7	Pipeline([(”seq”, Pipeline([(”VarianceThreshold”, VarianceThreshold()), (”MinMaxScaler”, MinMaxScaler())])), (”LogisticRegression”, LogisticRegression())])	0.875	0.02
8	Pipeline([(”DecisionTree”, DecisionTreeClassifier())])	0.96875	7.49
9	Pipeline([(”seq”, Pipeline([(”Recursive Feature Elimination”, RFE(LinearSVC())), (”MaxAbsScaler”, MaxAbsScaler())])), (”DecisionTree”, DecisionTreeClassifier())])	0.96875	7.17
10	Pipeline([(”MinMaxScaler”, MinMaxScaler()), (”LogisticRegression”, LogisticRegression())])	0.90625	6.62

Table 11: Results of BFS runs on the ”seeds” dataset

Run	Found Pipeline	Test Accuracy	Duration (seconds)
1	Pipeline([(”LogisticRegression”, LogisticRegression())])	0.9418604651162791	14.83
2	Pipeline([(”MaxAbsScaler”, MaxAbsScaler()), (”LogisticRegression”, LogisticRegression())])	0.9302325581395349	10.41
3	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9883720930232558	12.14
4	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9883720930232558	13.53
5	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9651162790697675	14.35
6	Pipeline([(”VarianceThreshold”, VarianceThreshold()), (”DecisionTree”, DecisionTreeClassifier())])	0.9418604651162791	12.81
7	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9767441860465116	14.22
8	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9651162790697675	14.08
9	Pipeline([(”seq”, Pipeline([(”SelectPercentile”, SelectPercentile()), (”Binarizer”, Binarizer())])), (”RandomForest”, RandomForestClassifier())])	0.5465116279069767	12.08
10	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9534883720930233	0.51

Table 12: Results of BFS(mapd=2) runs on the ”wdbc” dataset

Run	Found Pipeline	Test Accuracy	Duration (seconds)
1	Pipeline([(”LogisticRegression”, LogisticRegression())])	0.98124191461837	361.63
2	Pipeline([(”RandomForest”, RandomForestClassifier())])	0.9793014230271668	359.77
3	Pipeline([(”StandardScaler”, StandardScaler()), (”LogisticRegression”, LogisticRegression())])	0.9831824062095731	361.13
4	Pipeline([(”MaxAbsScaler”, MaxAbsScaler()), (”LogisticRegression”, LogisticRegression())])	0.9799482535575679	366.4
5	Pipeline([(”StandardScaler”, StandardScaler()), (”Gradient Boosting Classifier”, GradientBoostingClassifier())])	0.9909443725743855	368.88
6	Pipeline([(”MinMaxScaler”, MinMaxScaler()), (”LogisticRegression”, LogisticRegression())])	0.9741267787839586	158.94
7	Pipeline([(”MaxAbsScaler”, MaxAbsScaler()), (”LogisticRegression”, LogisticRegression())])	0.9864165588615783	160.07
8	Pipeline([(”LogisticRegression”, LogisticRegression())])	0.981888745148771	159.78
9	Pipeline([(”MaxAbsScaler”, MaxAbsScaler()), (”LogisticRegression”, LogisticRegression())])	0.9799482535575679	159.67
10	Pipeline([(”StandardScaler”, StandardScaler()), (”LogisticRegression”, LogisticRegression())])	0.9864165588615783	159.42

Table 13: Results of BFS runs on the ”har” dataset