Solving ML with ML: Effectiveness of the Metropolis-Hastings algorithm for synthesizing Machine Learning Pipelines

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

Abstract

In AutoML, the search space of possible pipelines is often large and multidimensional. This makes it very important to use an efficient search algorithm. We measure the effectiveness of the Metropolis-Hastings algorithm (M-H) in a pipeline synthesis framework, when the search space is described by a context-free grammar. We also compare the performance of the M-H algorithm to other search algorithms. While AutoML frameworks use many different search algorithms, and comparisons between AutoML frameworks exist, this is the first paper that compares the performance of different search algorithms in the context of pipeline synthesis under equal conditions. We found that M-H is slightly outperformed by BFS2, the simplest possible search algorithm. We conclude that the datasets we use for evaluating the algorithms are too simple to meaningfully compare the performance of different search algorithms. We also conclude that for simple datasets, simple search algorithms work best.

1 Introduction

Setting up a machine learning (ML) pipeline can take up a lot of time. There are a myriad of ways to combine the preprocessing and feature selection steps with a classifier, and if you want to also consider ensembles of classifiers, the amount of possible pipelines becomes even larger. It is not always clear in advance which combinations of steps will work well together for a given problem, and sometimes different combinations need to be tried out. The process of trying out different pipelines and tweaking them can take a lot of time for data engineers, and even more for novices. So if the generation of ML pipelines could be done automatically, it could save experts some time and make machine learning more accessible to beginners.

An important challenge for automatically generating machine learning pipelines is that the search space of possible pipelines is huge. This means that having an efficient search algorithm is crucial for automating the pipeline search. In this paper we create an approach to program synthesis that uses a context-free grammar (CFG) to describe the search space of possible pipelines and measure the effectiveness of the Metropolis-Hastings algorithm [1] for searching through it. We also compare the effectiveness of the Metropolis-Hastings algorithm to other search algorithms. Multiple tools already exist for generating a machine learning pipeline for a given problem using a CFG to describe the search space [2][3], but no research has yet been done on which search algorithm works best for traversing the search space under equal conditions. In this paper we measure the performance of the Metropolis-Hastings algorithm in a pipeline synthesis framework, and compare it to other search algorithms.

The rest of this paper is structured as follows. First, section 2 examines related literature and points out what this paper aims to contribute to the literature. Then, section 3 describes the methodology. It contains a description of our implementation of the M-H algorithm, how we represent the search space, and how we evaluate and compare the performance of different search algorithms. After that, section 4 discusses the details of how the grammar is set up, and shows which parameters are used for the M-H algorithm. The results are shown in section 5 and subsequently discussed in section 6. This is followed by a reflection on the reproducibility and integrity of performed research in section 7. And finally, section 8 summarizes the main findings and contains suggestions for further work.

2 Literature overview

The field of automated machine learning (AutoML) has multiple problems, which include neural architecture search, hyperparameter optimization, Combined Algorithm Selection and Hyperparameter tuning (CASH), and pipeline synthesis [4]. In this section we focus only on pipeline synthesis and the CASH problem, as those are most relevant to our research.

An important challenge in both of these problems is that the search space of possible pipelines is very large and multidimensional [5], which makes it difficult to find a good solution. AutoML frameworks have tried to solve this problem with a number of different approaches.

First of all, many different search algorithms and optimisation techniques have been tried to traverse the search space more efficiently. One of the earliest AutoML frameworks was Auto-WEKA [3], which used Bayesian optimisation to select the best pipeline for a given ML problem. Later, TPOT achieved better performance by using an evolutionary algorithm [6]. Auto-sklearn added meta-learning to Bayesian optimisation to achieve even better performance and win an international AutoML competition [2]. SmartML also achieves good performance, while using only meta-learning [7]. Autostacker uses an evolutionary algorithm in combination with hierarchical stacking to find the best pipeline. Monte Carlo Tree Search (MCTS) was used by the Mosaic framework to achieve state of the art results [8]. Another framework translates the problem into a classical planning model, and then uses existing planners to search for the best pipeline [9]. And finally, AlphaD3M coupled meta reinforcement learning with MCTS to achieve a similar performance as TPOT and AutoSKLearn while being an order of magnitude faster [10].

Another approach is to change the representation of the search space. RECIPE uses a genetic algorithm and defines the search space with a context-free grammar (CFG) [11]. This ensures that only valid pipelines can be generated, which makes the search space smaller and improves performance. Similarly, the performance of AlphaD3M was improved by representing the search space as a CFG [5].

When the search space is defined with a grammar, the shape of permitted pipelines can also be changed. On one hand, some frameworks require the pipelines to consist of a predefined amount pipeline steps put in parallel [12]. On the other hand, the PIPER and RECIPE frameworks allow pipelines to be in the shape of any Directed Acyclic Graph (DAG) [13][11]. While this significantly increases the size of the search space, it potentially also adds a lot of useful pipelines that have a more complex structure.

Finally, the ML primitives that are used can also be changed. Many popular frameworks use scikit-learn [14] or weka [15], but even when frameworks use the same ML library, they often use a different subset of the ML operators.

A number of studies have compared the overall performance of different AutoML frameworks [16][17]. But no comparison has yet been done of the performance of different search algorithms under equal conditions. This paper aims to fill part of this gap by comparing the performance of seven search algorithms on the task of pipeline synthesis by searching through the same search space defined by a CFG.

3 Methodology

The methodology section consists of five parts. The first part describes the benchmark of machine learning problems we created to measure the performance of different search algorithms. Then, the second part explains how the grammar that describes the search space is implemented. The third part discusses how the quality of a pipeline is evaluated. After that, the fourth part describes our implementation of the M-H algorithm. And the final part explains how we compare the performance of different search algorithms.

3.1 Dataset of ML problems

We first assemble a dataset of machine learning problems. This dataset can be used as a benchmark to compare the performance of different search algorithms on the task of pipeline synthesis.

To limit the scope of this project, we focus solely on multi-class classification problems where all features have binary, numerical or categorical values. We gathered our datasets from the OpenML repository [18].

The dataset contains a wide variety of ML datasets to ensure that different types of ML problems are represented in the benchmark. The datasets are divided into three categories: simple, complex and from related research [19][6][20]. The datasets also have different sizes: the smallest has 150 entries, and the largest 13.9k entries. More information about the datasets in the benchmark can be found in Table 1.

Unfortunately, we could not utilize enough computational resources to evaluate the search algorithms on the entire benchmark, so the experiments are performed on five datasets from the benchmark. According to AutoML best practices, the datasets used for setting the parameters should be different from those used for evaluation [21]. So two datasets are used to select the best parameters for the M-H algorithm, and three other datasets are used to compare the performance of the different search algorithms. Both the parameter selection datasets and evaluation datasets contain datasets of varying sizes to ensure that the search algorithms are tested on different amounts of data. The datasets that are used for parameter selection and evaluation can be seen in Table 2.

Since accuracy is one of the most popular and straightforward metrics, we use it as our metric for the quality of pipelines. And because none of the datasets are unbalanced, accuracy can be used as metric [22].

(a) Simple datasets

Name	ID	Entries	Features	Target classes
iris	61	150	4	3
seeds	1499	210	7	3
blood-transfusion	1464	748	4	2
diabetes	37	768	8	2
ilpd	1480	583	10	2
qsar-biodeg	1494	1.1k	41	2
monks-problems-2	334	601	6	2
tic-tac-toe	50	958	9	2

(b) Complex datasets

Name	ID	Entries	Features	Target classes
gas-drift	1476	13.9k	128	6
musk	1116	6.6k	167	2
madelon	1485	2.6k	500	2
gisette	41026	7.0k	5.0k	2
har	1478	10.3k	561	6

(c) Datasets from related research

Name	ID	Entries	Features	Target classes
glass	41	214	9	6
car-evaluation	40664	1.7k	21	4
wdbc	1510	569	30	2
spambase	44	4.6k	57	2
wine-quality-red	40691	1.6k	11	6
wine-quality-white	40498	4.9k	11	7

Table 1: Datasets included in our benchmark for measuring the performance of AutoML tools.

(a) Datasets for parameter selection

Name	ID	Entries	Features	Samples per class
diabetes	37	768	2	268-500
spambase	44	4.6k	2	1813-2788

(b) Datasets for evaluation

Name	ID	Entries	Features	Samples per class
seeds	1499	210	3	70-70
wdbc	1510	569	2	212-357
har	1478	10.3k	6	1.4k-1.9k

Table 2: Datasets for parameter selection and evaluation. The last column shows the size of the smallest and largest class. Since in none of the datasets the largest class is significantly larger than the smallest class, accuracy can be used as a metric for performance.

3.2 Defining the search space

This section describes the grammar we use for representing the search space. A good grammar needs to find a balance between being expressive enough to contain useful pipelines, and being restrictive enough to allow for efficient search [23].

As a starting point for the grammar, we take the grammar proposed by a related research paper [9]. Unlike many other AutoML frameworks, it allows pipelines to be in the form of a directed acyclic graph (DAG). This means that pipelines elements can be placed in sequence or in parallel in any combination, which increases the amount of useful pipelines that can be made.

The structure of this grammar has three rules. The first rule states that every pipeline ends with a classifier. The second rule states that two steps can be put in sequence one after another. And the third rule states that two steps can be put in parallel. When two operations are put in parallel, the outputs of the steps are concatenated and used as input for the step that comes afterwards. Figure 1 shows the rules as well as a more complex example of a pipeline generated by the grammar. To fill in the grammar, we use three types of terminals: preprocessing operators, feature selection operators, and classifiers. These are described in section 4.2.

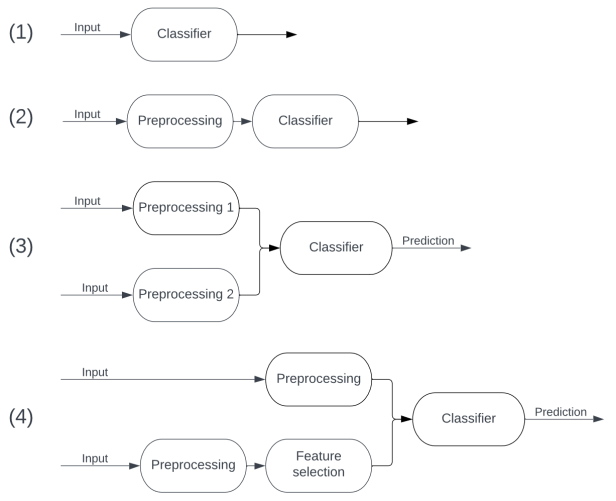

Figure 1: Examples of pipelines that the grammar allows. (1) is a pipeline with only a classifier. (2) shows two pipeline steps in sequence. (3) shows two steps in parallel. The input is copied for the parallel steps, and the output of both steps is concatenated and given as input for the next step. Finally, (4) shows a more complex pipeline that can be generated by the grammar. In these examples all preprocessing steps can be replaced with a feature selection step and vice versa.

Setting up the structure of the grammar as shown in Figure 1 gives us a lot of flexibility in how to combine the pipeline steps. But it also allows for pipelines that we know in advance are not going to work. Thus, we change the structure of the grammar in two ways to eliminate some of the useless pipelines and reduce the search space.

First, we remove the NoOp() operation, which is one of the terminals in the original grammar [9]. This operation does nothing and passes the input it gets on to the next operation without changing it. The new grammar does not have NoOp() and is equally expressive, but has a smaller search space, allowing for faster search.

Secondly, the original grammar allows classifiers to be put in such a way that their prediction is the only input for the next pipeline step. These pipelines do not make sense because the second part of the pipeline only gets one prediction as input and no other information to improve that prediction. This means that the second part of such a pipeline is useless. To combat this, our grammar does not allow classifiers in such compositions that their output is the only input for the next pipeline step. This reduces the search space and does not eliminate any reasonable pipelines.

3.3 Evaluating pipelines

To ensure that the results are representative, we split the datasets into a train, validation and test set in a 70-15-15 split. When a search algorithm selects a candidate pipeline, the pipeline is trained on the train set. Then, the performance of the pipeline is tested on the validation set. After considering many pipelines, the search algorithm selects the pipeline that scored best on the validation test. Next, we measure the performance of the pipeline on the test set. The test set is never seen before by the search algorithm and provides an unbiased indication of the quality of a pipeline. This approach is in line with ML best practices and mirrors what many AutoML frameworks do [2][6].

But since we do could not utilize larger computational resources, we speed up the evaluation by training each pipeline on only a part of the train set. We do this by randomly selecting 300 pipelines from the train set and training a pipeline on those. This significantly speeds up the evaluation process, as training a pipeline on 70 percent of a large dataset can take a long time. We still test the pipelines on the full test set, to reduce variability in the results we find.

3.4 Metropolis-Hastings algorithm

Our implementation of the M-H algorithm works as follows. We start with a random pipeline. Then, for each iteration t, we choose a candidate pipeline from the neighborhood of the current pipeline. If the candidate pipeline produces a better accuracy than the current pipeline, the candidate pipeline becomes the new current pipeline. If the accuracy of the candidate pipeline is lower than the current pipeline, there is still a chance that the candidate pipeline will become the new current pipeline. This chance is proportional to the accuracies of the two pipelines. During each iteration, we also keep track of the best pipeline and return it at the end. The pseudocode for our implementation of the algorithm is given below.

We use Herb.jl[1] to get the neighborhood of a pipeline, and choose a random neighbor in line 4. In line 6, rand(0,1) returns a uniform random number between 0 and 1.

3.5 Evaluating the search algorithm

To make the comparison between search algorithms fair, we evaluate each search algorithm on the same problems, use the

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

Algorithm 1 Metropolis-Hastings algorithm

```
 1: $current \leftarrow$ random pipeline
 2: $best \leftarrow current$
 3: while iteration $t < max\_iterations$ do
 4:     $candidate \leftarrow$ random neighbor of $current$
 5:     $a \leftarrow accuracy(candidate)/accuracy(current)$
 6:     if $a > rand(0, 1)$ then
 7:         $current \leftarrow candidate$
 8:         if $accuracy(current) > accuracy(best)$ then
 9:             $best \leftarrow current$
10:         end if
11:     end if
12: end while
13: return $best$
```

same grammar, and allow each search algorithm to consider only 100 pipelines.

We compare the results of the Metropolis-Hastings algorithm (M-H) to the results of the following search algorithms: very large-scale neighborhood search (VLSN) [24], a genetic algorithm (GA) [25], Monte Carlo tree search (MCTS) [26], A* [27], and two versions of breadth-first search (BFS2 and BFS4).

BFS2 is breadth-first search with enumeration depth 2, and BFS4 has enumeration depth 4. The enumeration depth determines the maximal amount of steps that may be taken in the grammar to arrive at a derivation. With enumeration depth 2, BFS2 can only construct the five pipelines consisting of a classifier and nothing else. BFS4 can derive many more pipelines, but for the comparison we only look at the first 100.

Like many of the search algorithms that are compared in this paper, the M-H algorithm is stochastic and does not find the same pipelines every time. Because of this, the algorithm sometimes finds much better solutions than other times. To combat this variability, we run each search algorithm ten times on each problem and calculate the mean accuracy and the sample variance for each algorithm.

4 Experimental setup

This section discusses how the grammar is implemented, the exact machine learning operators that are used, and which parameters are selected for the M-H algorithm.

4.1 Context-free grammar

We use the library Herb.jl[2] to define the grammar. The resulting pipelines are made in scikit-learn [14], which makes it very easy to create the pipelines, since scikit-learn already has functions for putting two pipeline steps in sequence or in parallel. All of the pipeline steps are also operators from scikit-learn. The structure of the grammar is shown in Figure 2. Section 4.2 describes the machine learning operators we use as terminals for the grammar.

To make our grammar work with Pipeline and FeatureUnion from scikit-learn, each terminal is a tuple of a name and a function, and we append an increasing number to each of the pipeline step names. For example, when the following pipeline is generated by the grammar,

```
Pipeline([("PCA", PCA()), ("XGBoost", XGBoost())])
```

it is changed into the pipeline below to guarantee that each pipeline step has a different name.

```
Pipeline([("PCA1", PCA()), ("XGBoost2", XGBoost())])
```

This is especially relevant for combining two Pipelines or FeatureUnions, as otherwise they would both have the name "seq" or "par", and the resulting pipeline would fail.

```
<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 2: Our CFG describing the search space of possible pipelines. The variables <preproc>, <fselect> and <classif> evaluate to one preprocessing step, feature selection step, or classifier respectively. Section 4.2 describes which pipeline operators we use as terminals for the grammar.

4.2 Machine learning operators

The terminals in our grammar consist of three kinds of machine learning operators: preprocessing steps, feature selection steps, and classifiers. For the terminals, we use the same 17 ML operations that are used by TPOT [6], with the exception that we use GradientBoostingClassifier instead of XGBoost. Because of this change, all ML operators are from scikit-learn. The exact operators used in the grammar are shown below.

Preprocessing steps: StandardScaler, RobustScaler, MinMaxScaler, MaxAbsScaler, PCA, Binarizer and PolynomialFeatures.

Feature selection steps: VarianceThreshold, SelectKBest, SelectPercentile, SelectFWE and RFE.

Classifiers: DecisionTreeClassifier, RandomForestClassifier, GradientBoostingClassifier, LogisticRegression and KNeighborsClassifier.

Hyperparameter optimization is not in the scope of this paper, so we use the default parameters for all operators except SelectKBest and RFE, which don't have default parameters. For SelectKBest we use k=4, and for RFE we use a LinearSVC as optimizer, since scikit-learn recommends this optimizer for the type of datasets we are using.[3]

4.3 Parameter selection

The algorithm has two parameters that can be optimized: enumeration_depth and max_pipeline_depth. Which parameters

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

[3]https://scikit-learn.org/stable/tutorial/machine_learning_map

work best partly depends on the size of the dataset. To account for this, we test the parameters on two datasets of different sizes.

The enumeration depth determines the maximal depth of the pipeline part that is generated to replace part of the current pipeline. If this depth is too small, all candidate pipelines will be very close to the current pipeline, and the algorithm may get stuck in a local maximum. If the enumeration depth is too large, the search algorithm may become very slow and the candidate pipelines may be very different from the current pipeline, because of which the search algorithm may fail to converge.

The maximal pipeline depth determines the maximal depth of any candidate pipeline. If the max_pipeline_depth is too small, only simple pipelines are considered and the algorithm may not find a good pipeline. If it is too large, the algorithm may find an unnecessarily complex pipeline or keep increasing the complexity of a pipeline instead of trying other simple pipelines.

Figure 3 shows the performance of the algorithm with different parameters on the diabetes and spambase datasets.

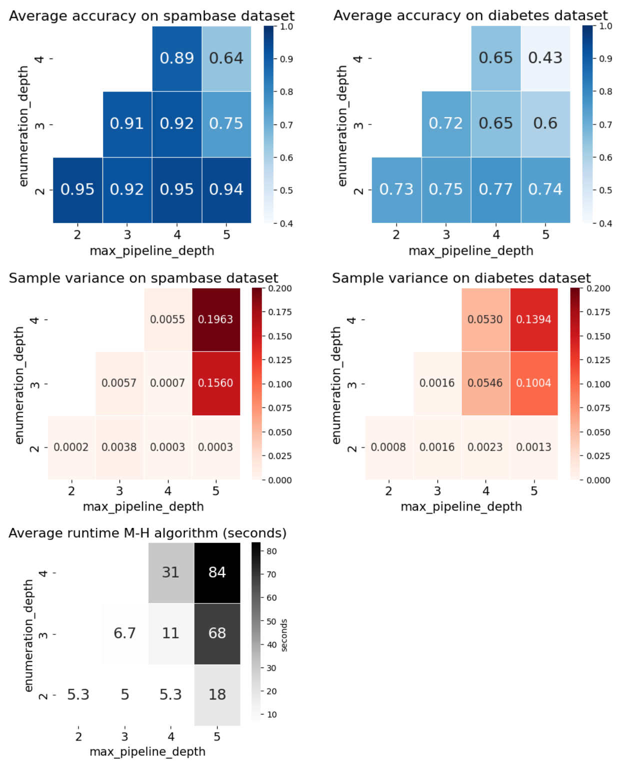

Figure 3: The two upper plots show the average accuracy of the pipelines found by the M-H algorithm with different parameters. The two middle plots show the sample variance in the accuracy. And the lowest plot shows the average runtime of the algorithm in seconds. The algorithm performs significantly better with enumeration_depth=2 than with other enumeration depths. enumeration_depth=2 with max_pipeline_depth=4 provides the best accuracy. Although, all settings with enumeration_depth=2 work well and the difference between them is small. Also, the runtime greatly increases when the values for the parameters become large.

(a) Average accuracy

Algorithm	seeds	wdbc	har
BFS2	0.931	0.969	0.980
BFS4	0.925	0.949	0.982
M-H	0.919	0.959	0.969
VLSN	0.906	0.949	0.980
GA	0.847	0.912	0.760
MCTS	0.928	0.970	0.981
A*	0.919	0.965	0.970

(b) Sample variance

Algorithm	seeds	wdbc	har
BFS2	0.00147	0.00027	0.00002
BFS4	0.00221	0.00091	0.00002
M-H	0.00265	0.00052	0.00045
VLSN	0.00239	0.00043	0.00003
GA	0.03352	0.01569	0.16150
MCTS	0.00110	0.00034	0.00001
A*	0.00178	0.00054	0.00054

Table 3: Average accuracy and sample variance of the different search algorithms on the evaluation datasets. The differences in the accuracies are very small. BFS2 and MCTS have the best performance, GA the worst, and the rest perform similiarly to each other.

The algorithm performs best for enumeration_depth=2, and worst when both parameters have large values. On both datasets, the algorithm gets the best results with enumeration_depth=2 and max_pipeline_depth=4, so we use these values when we compare the M-H algorithm to other search algorithms.

The value of the parameters also influences the runtime of the algorithm. When the values are small, the algorithm runs fast. When the values become large, the runtime increases greatly. This can be relevant for practical applications, as a faster runtime can be important to users.

5 Results

We evaluate the performance of each search algorithm by running it on three evaluation datasets and allowing each algorithm to train only 100 pipelines. We measure the performance of each search algorithm ten times on each dataset and calculate the mean and the sample variation. The results are shown in Table 3.

The most noticeable result is that none of the complicated search algorithms significantly outperform the simple BFS2. Also, the accuracy is quite high on all datasets. On the seeds dataset most algorithms achieve an accuracy over 0.9. On the wdbc dataset, an accuracy of 0.95 is achieved by most algorithms. And on the har dataset, most algorithms achieve 0.97 accuracy.

We can also see that the accuracy of an algorithm is related to its sample variance. The GA has the worst accuracy on all of the datasets, and its sample variance is also the highest. MCTS achieves the highest accuracy on the wdbc and har datasets, and has the lowest sample variance. But this relation is not perfect, as MCTS also has the lowest sample variance on the seeds dataset, but BFS2 has a better accuracy.

When we look at the average accuracy, the best algorithm seems to be either BFS2 or MCTS, as MCTS has 0.001 better accuracy on the wdbc and har datasets, while BFS2 has 0.003 better accuracy on the seeds dataset. It should be noted however that the differences are very small, and may just be due to random variation.

We also see that BFS2 gets better results than all other algorithms except for MCTS, which achieves very similar results. An interesting observation is that BFS2 also scores better than BFS4, even though all pipelines considered by BFS2 are also considered by BFS4.

The average accuracy of the M-H algorithm is about 0.01 lower than BFS2 and MCTS on all three datasets. When we compare M-H with BFS4, BFS4 scores 0.006 better on the seeds dataset, M-H scores 0.01 better on wdbc, and BFS4 scores 0.013 better on the har dataset. M-H achieves an accuracy that is about 0.01 higher than VLSN on the two smaller datasets, and on the larger dataset, but on the larger dataset it is the other way around. M-H and A* have very similar average accuracies, but A* performs 0.006 and 0.001 better on the two larger datasets. And like all other search algorithms, M-H significantly outperforms GA.

The only two algorithms that seem to perform better than the rest are BFS2 and MCTS. GA scores much worse than the other algorithms on all three datasets. The difference between the other algorithms is very small, and random variation probably contributed to these differences. So we can say that BFS2 and MCTS perform best, GA performs worst, and the rest have approximately the same performance. And surprisingly, none of the more complex algorithms achieve better results than the simple BFS2.

6 Discussion

The most striking result is that none of the search algorithms outperform BFS2, since BFS2 only tries the five classifiers on their own and nothing else. This section examines possible explanations for why no search algorithm outperforms BFS2, and specifically for why M-H does not outperform BFS2.

BFS2 is most easily comparable to BFS4, since BFS4 not only tries the five pipelines BFS2 tries, but also many more. It may therefore seem counterintuitive that BFS4 performs worse than BFS2, while trying more pipelines. But we believe that the difference in performance can be explained by random variation in results on the validation and test sets.

Evaluating a search algorithm happens as follows. Each search algorithm goes through the pipelines in some order, trains them, and selects the pipeline that performs best on the validation set. Then, the accuracy of that pipeline is evaluated on the test set. This is done ten times, and an average of the results on the test set is taken.

After BFS4 tries the five simple pipelines from BFS2, it tries 95 more complex pipelines. Even if these complex pipelines would be slightly worse than the simple pipelines, there would still be a chance that one of the complex pipelines would happen to do well on the validation set and get chosen for evaluation. Since the complex pipeline is slightly worse than the simple pipelines (but happens to do better on the validation set), it gets slightly worse results on the test set on average. This is one possible explanation for why the algorithm that tries more pipelines produces worse results.

Another possible explanation would be that (part of) the difference between BFS2 and the other search algorithms is due to BFS2 having gotten lucky. There is a slight variation in the results of a pipeline on the test set. If a search algorithm selects the same pipeline multiple times on the same dataset, it may get slightly different results. This is partly because each time a pipeline is evaluated, it sees a different part of the dataset and thus achieves a slightly different accuracy. So the results may also be (partly) explained by that BFS2 just got lucky with how it evaluated on the test set.

It is difficult to tell how much of the difference in performance is due to random variation, and how much due to some other explanation. The difference in average accuracy is 0.02 on the wdbc dataset and less than 0.006 on the other datasets, so it seems like random variation may be the most important reason for why BFS2 performed better than BFS4.

The other search algorithms (except GA) perform approximately equally well as BFS4. So if the difference in performance is because of random variation, we could conclude that there is no significant difference in performance between any of the search algorithms (except GA). And if the difference in performance between BFS2 and BFS4 is not fully explained by random variation, we could conclude that limiting the search to only very simple pipelines improves the performance on the evaluation datasets.

So the possible conclusions are either that for the evaluation datasets, very simple pipelines perform better than more complex pipelines, or that the complexity of a pipeline doesn't really matter, and that the better performance of BFS2 is due to random variation and nothing else.

In any case, the main conclusion is that the evaluation datasets are too simple to correctly assess the performance of the search algorithms. As stated in the introduction, one of the challenges in AutoML is that the search space of possible pipelines can be huge, which makes it important to have a good search algorithm. But for the evaluation datasets, the five classifiers on their own performed as good as any complex pipeline, and therefore the search space of useful pipelines has size five. With a search space this small, the best approach is obviously to just try all options (which is what BFS2 does). And any search algorithm that is more complicated than this is unnecessary. To truly measure the performance of the different search algorithms, we need to compare them on more complicated datasets.

This conclusion is the key to answering most of our research questions. The first research question is about how to create a dataset that is representative of general ML problems. We cannot say whether the benchmark proposed in section 3.1 is representative, since it is not used in the experiment, but we can conclude that the evaluation datasets are not representative of general ML problems, because they only include simple datasets.

The second question is about how to create a grammar for assembling ML pipelines that strikes the right balance between expressiveness and restrictiveness. The results show that which grammar is optimal depends on the datasets. If as is the case in our experiment, the datasets are very simple, a very simple grammar will do best. A grammar that only allows one classifier and nothing else would have produced the same results as BFS2, and it would not have allowed a search algorithm to waste time on more complex pipelines. But when the datasets are more complicated, we believe a grammar that also allows preprocessing and feature selection steps will do better, because for many real datasets pipelines that include preprocessing and feature selection steps do better than a classifier on its own. Since the grammar was not tested on complex datasets, we cannot conclude anything about whether a grammar should generate only pipelines with a fixed number of steps in sequence, or whether grammars should allow pipelines of many different forms.

The next research question asks how to measure the quality of a pipeline. This also depends on the datasets used, since accuracy is a good metric for balanced datasets, but for unbalanced datasets another metric is needed.

The final research question asks which search algorithms perform well under which circumstances. The results show that on very simple datasets BFS2 performs best. And since all evaluation datasets are simple, we cannot conclude anything about which algorithms do well on complex datasets. Similarly, we know that for very simple datasets it is better to use BFS2 than M-H, but we cannot say anything about the performance of M-H on more complex datasets.

To conclude the discussion, we would like to note that this paper uses a simple implementation of the M-H algorithm. Improving the implementation of the algorithm may lead to an increase in performance. Although, we doubt an improved version of the M-H algorithm would do much better on the evaluation datasets, as on those datasets none of the more complex search algorithms outperforms the simple BFS2.

7 Responsible research

For this project, we use publicly available datasets that do not have personal information. The datasets are taken from the OpenML[4] repository. We believe this is a good choice, as this is a reputable repository and many papers that compare AutoML frameworks get their datasets from OpenML. The topics we are working on are also not related to criminal or unethical topics. So reproducibility and credibility are the most important parts of the discussion about responsible research for this project. They are covered in the following sections.

7.1 Reproducibility

To ensure our results are reproducible, we host the full codebase on the TU Delft repository[5]. The code is extensively documented and can easily be cloned and executed to check our results. The experiment can be replicated using a normal laptop, as the runtime should not exceed a couple of hours per algorithm. To evaluate a search algorithm on the full benchmark provided in section 3.1, access to larger compute resources might be needed. The datastets are also publicly available on OpenML[6] and our code includes a function to download the relevant datasets. To allow fellow researchers to further check our results, we also host the logs of all of the runs of the experiment. Each of these logs shows information about one run the the experiment. It shows which search algorithm was used on which datasets, and which accuracy it achieved. The exact parameters, runtimes and results are also documented in the logs. This allows anyone to see not only the aggregate statistics, but also the exact results of each run.

7.2 Credibility

Unfortunately, we could not utilize larger computational resources during this experiment. Because of that, only three datasets were used for testing the datasets. As was said in the discussion, this causes potential issues with the credibility and generalisability of the results.

First of all, the sample size is very small. Secondly, the evaluation datasets themselves are relatively small; the largest evaluation dataset has 10k features, while in real life some datasets have multiple millions of entries. Thirdly, all evaluation datasets are balanced (each target class has approximately the same size). This is not the case in many datasets in the real world, as often some target classes have much fewer entries than other target classes. And finally, as stated in the discussion, the evaluation datasets turned out to be very simple. These factors make the evaluation datasets less representative of general ML problems, and if the experiment is done on different datasets, different results may be found.

A potential solution to improve the credibility of the results would be to run the same experiment on the whole benchmark provided in section 3.1. This would make the results more generalisable, as the benchmark contains more datasets, some of which are large, unbalanced or difficult.

And finally, as mentioned in the discussion, this paper compares specific implementations of each algorithm. It does not say anything definitive about which search algorithm is better for program synthesis. If one of the algorithms is implemented in a different way or if a different version of the same algorithm is implemented, we might find different results.

8 Conclusions and Future Work

The goal of this paper is to compare the performance of the Metropolis-Hastings algorithm to other search algorithms on the task of finding good pipelines in a CFG for a given ML problem. We find that for the evaluation datasets the best approach is to only try the five classifiers on their own and to avoid trying anything more complicated. This leads to both optimal accuracy and a much shorter runtime. We also conclude that to test the effectiveness of different search algorithms, we need to use datasets that are more difficult, so that the search space of good solutions is large.

An obvious direction for further research would be to repeat our experiment on the full dataset of ML problems we provided in section 3.1 and perhaps on even larger and more difficult benchmarks. It could also be interesting to research for which types of datasets complex pipelines work well, and for which very simple ones perform better.

[4]https://www.openml.org

[5]https://repository.tudelft.nl

[6]https://www.openml.org/

Another direction for further research could be to add more ML operators to the grammar. Our grammar covers a variety of basic ML operators, but there are many operators that are not included in the grammar, such as XGBoost. Including these operators in the grammar may improve the performance on certain types of datasets.

During the evaluation of the search algorithms, we noticed that some ML operators take much longer to train. On the har dataset pipelines with Recursive Feature Elimination or the Gradient Boosting Classifier take more than twenty times as long to train as pipelines without these operators. We think it would be interesting to compare the effectiveness of different search algorithms when they are not bounded by the amount of pipelines they can evaluate, but by their runtime. This would mean that the search algorithms would also have to consider the expected runtime of each pipeline step. Further research in this direction may be useful for practical applications because it could lead to significant decreases in the runtime of AutoML tools.

References

[1] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," 1970.

[2] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2015/file/11d0e6287202fced83f79975ec59a3a6-Paper.pdf

[3] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Combined selection and hyperparameter optimization of classification algorithms," ser. KDD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 847–855. [Online]. Available: https://doi.org/10.1145/2487575.2487629

[4] X. He, K. Zhao, and X. Chu, "Automl: A survey of the state-of-the-art," *Knowledge-Based Systems*, vol. 212, p. 106622, 2021.

[5] I. Drori, Y. Krishnamurthy, R. Lourenco, R. Rampin, K. Cho, C. Silva, and J. Freire, "Automatic machine learning by pipeline synthesis using model-based reinforcement learning and a grammar," *arXiv preprint arXiv:1905.10345*, 2019.

[6] R. S. Olson and J. H. Moore, "Tpot: A tree-based pipeline optimization tool for automating machine learning," in *Proceedings of the Workshop on Automatic Machine Learning*, ser. Proceedings of Machine Learning Research, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., vol. 64. New York, New York, USA: PMLR, 24 Jun 2016, pp. 66–74. [Online]. Available: https://proceedings.mlr.press/v64/olson_tpot_2016.html

[7] M. M. M. Z. A. Maher and S. Sakr, "Smartml: A meta learning-based framework for automated selection and hyperparameter tuning for machine learning algorithms," in *EDBT: 22nd international conference on extending database technology*, 2019.

[8] H. Rakotoarison, M. Schoenauer, and M. Sebag, "Automated machine learning with monte-carlo tree search," *arXiv preprint arXiv:1906.00170*, 2019.

[9] M. Katz, P. Ram, S. Sohrabi, and O. Udrea, "Exploring context-free languages via planning: The case for automating machine learning," in *Proceedings of the International Conference on Automated Planning and Scheduling*, vol. 30, 2020, pp. 403–411.

[10] I. Drori, Y. Krishnamurthy, R. Rampin, R. d. P. Lourenco, J. P. Ono, K. Cho, C. Silva, and J. Freire, "Alphad3m: Machine learning pipeline synthesis," *arXiv preprint arXiv:2111.02508*, 2021.

[11] A. G. de Sá, W. J. G. Pinto, L. O. V. Oliveira, and G. L. Pappa, "Recipe: a grammar-based framework for automatically evolving classification pipelines," in *Genetic Programming: 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings 20.* Springer, 2017, pp. 246–261.

[12] S. Liu, P. Ram, D. Vijaykeerthy, D. Bouneffouf, G. Bramble, H. Samulowitz, D. Wang, A. Conn, and A. Gray, "An admm based framework for automl pipeline configuration," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 04, 2020, pp. 4892–4899.

[13] R. Marinescu, A. Kishimoto, P. Ram, A. Rawat, M. Wistuba, P. P. Palmes, and A. Botea, "Searching for machine learning pipelines using a context-free grammar," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 10, 2021, pp. 8902–8911.

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[15] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[16] I. Guyon, I. Chaabane, H. J. Escalante, S. Escalera, D. Jajetic, J. R. Lloyd, N. Macià, B. Ray, L. Romaszko, M. Sebag, A. Statnikov, S. Treguer, and E. Viegas, "A brief review of the chalearn automl challenge: Any-time any-dataset learning without human intervention," in *Proceedings of the Workshop on Automatic Machine Learning*, ser. Proceedings of Machine Learning Research, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., vol. 64. New York, New York, USA: PMLR, 24 Jun 2016, pp. 21–30. [Online]. Available: https://proceedings.mlr.press/v64/guyon_review_2016.html

[17] M.-A. Zöller and M. F. Huber, "Benchmark and survey of automated machine learning frameworks," *Journal of artificial intelligence research*, vol. 70, pp. 409–472, 2021.

[18] J. Vanschoren, J. N. Van Rijn, B. Bischl, and L. Torgo, "Openml: networked science in machine learning," *ACM SIGKDD Explorations Newsletter*, vol. 15, no. 2, pp. 49–60, 2014.

[19] J. P. Cambronero and M. C. Rinard, "Al: autogenerating supervised learning programs," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[20] R. S. Olson, W. La Cava, P. Orzechowski, R. J. Urbanowicz, and J. H. Moore, "Pmlb: a large benchmark suite for machine learning evaluation and comparison," *BioData mining*, vol. 10, pp. 1–13, 2017.

[21] M. Milutinovic, B. Schoenfeld, D. Martinez-Garcia, S. Ray, S. Shah, and D. Yan, "On evaluation of automl systems," in *Proceedings of the ICML Workshop on Automatic Machine Learning*, vol. 2020, 2020.

[22] M. Grandini, E. Bagli, and G. Visani, "Metrics for multi-class classification: an overview," *arXiv preprint arXiv:2008.05756*, 2020.

[23] S. Gulwani, O. Polozov, and R. Singh, "Program syntesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, p. 9, 2017.

[24] A. Sonneveld, T. Hinnerichs, and S. 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*, 2023.

[25] M. Butenaerts, T. Hinnerichs, and S. Dumancic, "Genetic algorithm-based program synthesizer for the construction of machine learning pipelines," *TU Delft preprint: available from repository.tudelft.nl*, 2023.

[26] B. Filius, T. Hinnerichs, and S. 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.

[27] R. Lejeune, T. Hinnerichs, and S. Dumancic, "Solving ml with ml: Effectiveness of a star search for synthesizing machine learning pipelines," *TU Delft preprint: available from repository.tudelft.nl*, 2023.