Genetic Algorithm-based Program Synthesizer for the Construction of Machine Learning Pipelines

Mathieu Butenaerts
Supervisor(s): Sebastijan Dumančić, Tilman Hinnerichs
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: Mathieu Butenaerts
Final project course: CSE3000 Research Project
Thesis committee: Sebastijan Dumančić, Tilman Hinnerichs, David Tax

Abstract

Because of the growing presence of artificial intelligence, developers are looking for more efficient methods to construct machine learning algorithms. Program synthesizers allow us to produce algorithms consisting of scalers, feature selection and classifiers. Each of these pipelines is a potential solution to the given machine learning task. The goal of this synthesizer was to find the best-suited pipeline for the problem, with a genetic search algorithm. The structure of the pipelines makes it easy to implement the cross-over and mutation properties of a genetic algorithm, as the pipelines and different algorithms very much resemble chromosomes and genes. Experiments were designed to measure the accuracies and runtimes of the synthesizer with the intent to compare them to the results of other synthesizers based on different search algorithms. The comparisons made could prove whether machine learning synthesizers are a viable solution to the mentioned development problem.

1 Introduction

Machine learning is an ever-expanding field in terms of accuracy and practicality in the modern world. More and more applications use machine learning in their features. Researchers and developers produce more convoluted and compound algorithms to tackle more complex problems [10]. Yet, the production of these algorithms consists of many time-consuming tasks, next to the implementation and the design of the algorithm itself. Tasks, like choosing the most effective machine learning design, choosing accuracy measurement methods, fine-tuning multiple parameters, and randomly initializing variables require educated intuition, but they are also very much trial-and-error-based. Facilitating these processes could be beneficial to every developer in the field of machine learning. S. Gulwani, O. Polozov, and R. Singh [7] have researched program synthesis and used machine learning methods like enumerative search, constraint solving, and stochastic search to find the most suitable pipeline for a given problem. A pipeline is a sequence of concatenated algorithms, produced by a program synthesizer, that serves as a solution to a given computational problem. The research group tasked with this project, used the notion of program synthesis, to construct synthesizers that produce the most suitable machine learning algorithms for the given datasets. Each member of the team was given a specific search method to investigate. While S. Gulwani already discussed genetic algorithms as a viable search algorithm in program synthesis, the research in this paper was focused on this method.

The goal of the research was to measure the performance of the given search algorithms in program synthesizers and why they are suitable options in the field of machine learning development. The performances of the synthesizers were compared to each other and analyzed why certain problems are better solved by which synthesizer.

This paper mentions the methodology used, a description of the experimental setup along with the results and a mention of the responsible research methods that have been applied with the possible ethical implications of our findings. The research paper concludes with a discussion, final conclusions, and the mention of future work on the subject, to which this project has led too.

2 Methodology

The strategy used to tackle the research questions consists of multiple sections. The first objective was to define the structure and lim-

itations of the pipelines. Context-free grammars were the best approach since their way of describing languages allowed us to define the different algorithms in a pipeline and the general structure of the sequences. With the grammar, a search space was created in which the respective search algorithms searched for the best solution for the given problem. The performance of the algorithms has been evaluated with the same metrics and the same datasets, in order to conduct objective comparisons and analysis.

2.1 context-free grammar

The search space consisted of machine learning pipelines. A single machine learning pipeline is a sequence of algorithms where each output of an algorithm is the input for the following algorithm. The machine learning pipelines are combinations of preprocessing and feature selection algorithms followed by a classification algorithm. A single pipeline contains all the functionalities of a complete classification algorithm. The structure and properties that a valid pipeline must adhere to, was best described by a context-free grammar. A context-free grammar can produce sequences based on a set of given rules, variables, and terminals. Starting with a given start variable, the context-free grammar transforms the variable to concatenations of terminals and variables, based on the given rules. Once all the variables have been replaced, a concatenation of terminals remains, acting as an instance of the language described by the grammar. [16]

The functionalities of grammars were used to produce different combinations of pipelines [3]. The grammar was allowed to construct directed acyclical graphs of preprocessing and feature selection algorithms, in order to construct more complex pipelines for the search space, while still ensuring a classification algorithm ends the sequence [8].

The grammar is defined in the Julia programming language with the Herb.jl framework [13] and the Scikit-learn library. The Pipeline object allowed us to concatenate the transformers in a sequence while the FeatureUnion object unionized the features transformed by the previous scalers and feature selection algorithms. These two functionalities acted as the joints in the directed acyclical graphs, combining the different branches of preparation algorithms towards a single classifier.

The preprocessing algorithms focused on for this research project, were the standard scaler, robust scaler, min-max scaler, maximum absolute scaler, Principal component analysis, binarization, and polynomial features as preprocessing procedures. The feature selection algorithms used were variance threshold, K-best, percentile, family-wise, and recursive feature elimination. As for classification algorithms, decision trees, random forest, gradient boosting, logistic regression, and nearest neighbour were used. These algorithms are the most often used in machine learning and have been well documented and supported by Scikit-learn, making them the best algorithms for the grammar. A full description of the grammar is given in the appendix (appendix figure 1).

2.2 Genetic Algorithms

The search algorithm analyzes the accuracy of the given pipelines and produces the most suitable solution for the given problem that it could find, based on the features of a genetic algorithm. A genetic algorithm is a stochastic search algorithm where the search approach is based on characteristics of biological evolution [11]. In the space of candidate solutions, the best-performing pipelines that were found were used to create a new search space with better candidate solutions. The reproduction procedure is done by combining the components of different selected pipelines. To introduce some variation in the search space, a mutation could take place, with a certain probability, which could alter a specific component in the candidate.

A general skeleton was implemented on

the Herb.jl repository constructed by Jaap De Jonge. The skeleton had the general structure of a genetic algorithm. The implementation of the features that would construct the initial population, select the best candidates from the generation, produce the next generation and perform the mutations still had to be added.

2.2.1 Fitness Function

The assessment of the quality of the pipelines was done by executing the sequence of machine learning algorithms using the Scikit-learn functionalities. The percentage of correct predictions on the total number of predictions represented the accuracy of the pipeline.

In the selection process, the rank selection strategy has been implemented with the intent to choose a suitable subset of candidates that would reproduce the next generation [6]. The current generation of candidate solutions was ranked based on their performance. Each specimen was given a chance of being selected for reproduction relative to their fitness. These chances were calculated by dividing the respective accuracies by the general sum of the accuracies for the generation. This method of selection allowed less accurate candidates to pass for reproduction. The goal of this strategy was to mimic the selection process as it occurs in nature as accurately as possible. To make sure one candidate is not selected twice, it is removed from the population, once the candidate has been chosen. Afterward, the selection process was repeated with updated selection chances. Only half of the population was selected for reproduction.

Other selection strategies were considered as well. Tournament selection considers a randomly selected subspace of the population and selects the best candidate in that subspace. This method was mentioned in the same paper as rank selection and was a strategy that was considered for the selection process in the genetic algorithm. Yet, the additional value it brought to the entire search algorithm was not worth the increased runtime that this strategy would introduce.

2.2.2 Cross-Over Function

The cross-over function was used to construct the new generation of pipelines. Two sequences of genes, in this case, sequences of transformers, were split on the same spot after which the end of one sequence was connected to the start of the other sequence [18]. This procedure created new pipelines, which possibly contain the best features of both parent pipelines.

Since the pipelines were designed as directed acyclical graphs, the cross-over would be more unorthodox. A branch from one graph was randomly selected and exchanged by a branch from the other, resulting in new pipelines.

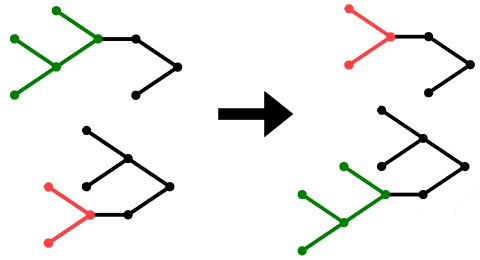

Figure 1: Example of cross-over operation

Since half of the previous population was eliminated in the natural selection section, each candidate would have to cross-over with two candidates. The population set was randomly ordered in a circular structure so each individual was adjacent to two other specimens. This individual would cross-over with their neighbors. With this method, it is ensured that each candidate reproduced with two unique other members of the population.

Randomly selecting branches from chromosomes and performing a cross-over, could result in pipelines that exceeded the maximum depth set in the initiation of the algorithm. In order to avoid this issue, branches

of the same depth were selected for cross-overs. This way both new sequences would remain the same size. This restriction reduced diversity in the search space, but its absence could cause the runtime of the algorithm to increase drastically because of the longer pipelines.

The reproduced pipelines could have an invalid structure that does not adhere to the defined grammar. A pipeline with too many feature selection algorithms would throw exceptions once it is executed. This could compromise the new generation and cause complications in future selection processes. Testing the validity of the pipeline after reproduction would show if it was a suitable candidate. If it proves to be valid, the new chromosomes were passed to the new generation. In any other case, the original parent chromosomes are reused in the next generation.

2.2.3 Mutation Function

The mutation function in a genetic algorithm is in charge of introducing minor changes to a selected candidate. The goal was to enforce diversity in the population and to ensure that the search space would not reach a dead end where cross-overs do not result in new combinations anymore [2].

In the context of the project, the mutation would change one algorithm in the pipeline with another. It was important to ensure that mutations change algorithms to algorithms of the same kind. Allowing a scaler to be changed to a classifier, would result in an invalid pipeline. In the implementation, a random algorithm in the pipeline is selected after which its type is looked up in the grammar. Afterward, a random algorithm was selected from the grammar of the same type that was different from the originally chosen algorithm that was to be replaced. After the algorithm in the pipeline was replaced by the new one, the mutation process was finished.

The mutations were executed on randomly selected candidates based on the mutation probability provided by the function call.

2.3 Evaluating The Performance

While analyzing the quality of the synthesizer, it was important to consider both the speed and accuracy of the best-produced pipeline, as well as the speed of the synthesizer itself. Since every member of the project group had been working on a synthesizer based on a different search algorithm, comparing the performances of the different strategies was fairly straightforward. From different experiments with different machine learning tasks, conclusions could be made from the findings about which strategies were better suited for which kind of problem. The performance of the overall search algorithm was related to the performance of a single pipeline which was indicated by its fitness value. The fitness value was the percentage of correct predictions over the entire testing data set. 70 percent of the data set was used as training data, 15 percent as testing data and 15 percent as validation data.

3 Experimental Setup and Results

To ensure the results were as objective as possible, the group agreed on certain parameters with the intention to run the respective algorithms in a similar environment. Each algorithm was run 10 times so unrepresentable coincidental results were eliminated. The number of times the fitness function was executed was also limited to 100. The depth of the grammar was in general set to 4 and the enumeration depth was 4. These parameters ensured stable conditions to run the algorithm on a personal computer with still a large search space. Other parameters more relevant to the respective algorithms were selected individually. The genetic algorithm has a mutation probability parameter which

was set to 0,1, implying that there was a chance of 10% that a mutation could occur in a chromosome. Since the population size was set to 10, there was a 65.13% that at least one mutation took place in the current generation. These parameters introduced a stable level of diversity to the algorithm. The dataset used in the development of the genetic algorithm was the iris dataset [5]. The small sample size allowed for quick evaluations and quick debugging performances.

The research group agreed on three datasets with which we would conduct the experiments, provided by the openML dataset database.

The seeds dataset [1] consists of 210 instances with 8 unique features and 3 unique classes. When running the genetic search algorithm, an average runtime of 12,94 seconds was measured. A notable observation was the variance in time which was 244,53. The quickest time was 0,97 seconds and the longest was 53,08 seconds. The average accuracy over the best pipelines found in each experiment was 0,847, implying that 84,7% of the instances in the test set were correctly classified. The best pipeline performed with an accuracy of 0,989.

The bwcd dataset [19] consists of 569 instances with 31 unique features and 2 unique classes. The genetic algorithm found pipelines with an average accuracy of 0,912 in an average time of 113,37 seconds. Yet again, the accuracy variance was very low (0,0157) and the runtime variance was very large (78114,791). The best pipeline for this dataset had an accuracy of 0,977.

The har dataset [12] consists of 10299 instances with 562 unique features and 8 unique classes. The pipelines classified the instances correctly with an average accuracy of 0,760 and a variance of 0,162. The best pipeline reached an accuracy of 0,988. The average runtime was 948,110 seconds with a variance of 1822586.

The other synthesizers that were analyzed during the research project were A* search [9], Monte Carlo Tree search [4], Metropolis-Hastings search [14] and Very Large Neighbourhood search [17]. Additionally, a breadth-first search algorithm was designed as a baseline with which the other algorithms could be compared. The algorithms were run on the same datasets with similar settings

The A* search algorithm was also run 10 times with a sample size of 300 and an enumeration depth of 5. The pipeline depth was set to 4. The average accuracies for the seeds, wdbc and har datasets were 0,919, 0,965 and 0,970 respectively. The variances were rather low with 0,00178, 0,000541 and 0,000538. The best pipelines for these datasets performed at an accuracy of 0,969, 0,988 and 0.984. The runtimes were on average 19,762, 32,12 and 1.220,96 seconds with variances 156,569, 321,35 and 1951733,27.

The Monte Carlo tree search ran 10 experiments as well with an enumeration depth of 3, a pipeline depth of 4 and a sample size of 300 instances. The iteration parameter, specific to this algorithm, was set to 100. For the seeds dataset, the synthesizer produced pipelines with an average of 0,928 and a variance of 0,00110. The runtime was on average 10,87 seconds with a variance of 2,81. The best pipeline, in this case, had an accuracy of 0,969. The pipelines found for the wdbc dataset could produce an accuracy of 0,970 on average with a variance of 0,000337. The best pipeline performed with an accuracy of 1,0. The average runtime was 166,04 seconds with a variance of 170910,42. Lastly, for the har dataset the Monte Carlo tree search algorithm found an average accuracy of 0,981 with a variance of 0,0000119. The runtime was on average 168,67 seconds with a variance of 691,867. The best pipeline for the har dataset performed with an accuracy of 0,988.

The Metropolis-Hastings search was performed on 1000 instances with an enumeration depth of 2 with a pipeline depth of 4. For the seeds dataset, the average accuracy found was 0,919. For the wdbc dataset, the accuracy was 0,920 and for the har dataset

it was 0,970. The respective average runtimes for the datasets were 6,19, 11,90 and 686,70 seconds. Again, the variance in accuracy over the 10 runs was rather low. Variances of 0,00156, 0,0176 and 0,000446 were found. In the case of runtime, larger fluctuations were observed with variances of 9,48, 17,81 and 24073,62. The best pipelines for these datasets ran with an accuracy of 0,967, 0,988 and 0,983.

The Very Large Neighborhood search ran 10 experiments as well, where the enumeration depth was set to 4 and the pipeline depth was 4. The sample space had a size of 300 and an additional parameter, maintaining the number of neighbours in every iteration was set to 25. The average accuracies corresponding to the experiments performed on the given datasets were 0,906, 0,949 and 0,979 with variances of 0,00239, 0,000427 and 0,0000298. The runtimes of these experiments were on average 8,67, 34,88 and 343,83 seconds. The runtime variances were 84,59, 342,33 and 28986,44 respectively to the datasets. The best pipelines produced for each dataset during the experiments had an accuracy of 1,0, 0,977 and 0,989.

For the baseline, a breadth-first search algorithm was used that would evaluate pipelines in order of increasing depth. 10 experiments were run on a sample size of 300 with an enumeration depth of 4 and a pipeline depth of 4. The average pipeline accuracies were 0,916, 0,949 and 0,982. The same accuracies had variances of 0,00348, 0,000907 and 0,0000212. The runtimes of these experiments were on average 12,58, 16,45 and 261,57 seconds with corresponding variances 0,481, 0,035 and 11565,20.

The statistics of the experiments are described in graphs in the appendix (appendix figure 2-39).

4 Responsible Research

In order to ensure the quality of the performed research, the origin of the used resources has to be explained as well as how the results from the mentioned experiments could be reproduced in order to prove their validity. Made conclusions should have merit and a sufficient scientific foundation to prove their validity.

4.1 Datasets

The resources that have been used should be verified and validated. The datasets used in the experiments and during the development of the search algorithms have been provided by OpenML. OpenML is a machine learning data set database that contained problems for many different languages like Julia, Python and Java. OpenML provides documentation for each dataset, mentioning the author, the study it was used for, and in which year the study took place as well as whether tr not they have been verified by OpenML themselves.

4.2 Reproducibility

The files with the code that produced the mentioned results were stored on a GitHub repository which has been referenced in the bibliography [15]. Clear documentation of the methods and structure has been provided in the files. Since the random generator has been provided with a constant seed, the results of all the search algorithms could be reproduced with the mentioned parameters, with the exception of the runtime result, which could have slight variations depending on the machine on which the algorithms were run.

4.3 Credibility

Every conclusion that has been made is supported by the mentioned results and statistics. It is important to consider that three arbitrary datasets have been used in the experiments. While their origin and their practicality concerning the project had been researched, they were used in the experiments without any prior preprocessing that might skew the results. The results and therefore

conclusions are unique to the datasets and no guarantees about the performance on other datasets can be made.

5 Discussion

The results have been recorded and presented in previous sections and the appendix. The first observation made was the low variances in the accuracies over the repeated experiments. The low variance implies that all the results are close together meaning the search algorithms repeatedly performed at a stable rate. The variance in runtime is rather large. Some of these variances can be explained by unregular spikes of runtime where the search algorithm took a long time of finding an optimal solution. Examples of this were the A* search algorithm on the har dataset (appendix figure 8), the Genetic algorithm on the wdbc dataset (appendix figure 18), the Genetic algorithm on the har dataset (appendix figure 20) and the Monte Carlo Tree search on the WDBC dataset (appendix figure 24). Other high variances were caused by irregular runtimes where the algorithm would run both fast and slow on the same experiment. This anomaly could be explained by particular combinations of machine learning algorithms in the pipelines that would take longer than others to be processed by the Scikit-learn framework.

Another particular occurrence in the findings is a 100% accuracy on the wdbc dataset by the Monte Carlo Tree Search. With a sample size of 300 instances and a 15% of testing data, the algorithm classified 45 of the 45 instances correctly. However unlikely, it would not be impossible to find a perfect accuracy.

Other irregularities were an accuracy of 0 on the HAR dataset by the Genetic algorithm search for two experiments and an accuracy of 0.547 for the Metropolis-Hastings search on the WDBC dataset. What makes these occurrences irregular is that they vary much from the mean of all the experiments done by the search algorithms on the particular datasets. This could be caused by unforeseen fallacies in the synthesizers that might have caused the constructed pipelines to be less accurate than desired.

The lowest accuracy found over all the experiments which was within a reasonable range of the mean of the related experiments, was by the very large neighborhood search with an accuracy of 0,84375 on the seeds dataset, most likely caused by the small sample spaces that the seeds dataset provides.

The best pipelines found for the mentioned datasets are a robust scaler followed by a logistic regression classifier for the wbcd dataset with accuracy 1,0 by the Monte Carlo Tree search and a pipeline consisting of a standard scaler followed by a gradient Boosting classifier for the har dataset with accuracy 0,99 by the breadth-first algorithm. For the seeds dataset, there was a four-way tie between the A* search, breadth-first search, genetic algorithm search, Monte Carlo Tree search and the Metropolis-Hastings search, all with an accuracy of 0,9688. The found pipelines were a pipeline with a standard scaler, a principal component analysis and a max absolute scaler with a logistic regression classifier, a pipeline with a standard scaler and a max absolute scaler with a logistic regression classifier, a pipeline with two robust scalers with a gradient boosting classifier and a pipeline with a random forest classifier, respectively to the mentioned synthesizers. The search algorithm that found their pipeline the fastest was the Metropolis-Hastings with 6,44 seconds. The short runtime corresponds to the short pipeline that was produced by the synthesizer since the algorithm would have noticed the high accuracy and would have stopped the search early since no improvements would have been registered.

6 Conclusion

The goal of this research was to examine whether synthesizers used for constructing pipelines of machine learning algorithms

were the solution to the time-consuming tasks that come with developing classifiers for given datasets. To prove this, experiments were designed where synthesizers based on different search algorithms tried to find the best pipeline for three given classification problems. The accuracies and runtimes were compared to each other and a baseline synthesizer consisting of a breadth-first algorithm. The results of the experiments concerning the accuracies were promising given that the variances were low and the averages were mostly above 90%. The runtime on the other hand fluctuated often, most likely caused by pipelines that were incompatible with the fit functions provided by the Scikit-learn framework. The best pipelines for the datasets were provided by the Metropolis-Hastings, Monte Carlo Tree Search and the Breadth-first search. Yet their performance was not much greater than the other synthesizers. In the case of the mentioned datasets, there is not a great notable difference in quality among the search algorithms. The simple baseline breadth-first search is as qualified to find quality pipelines as the other synthesizers. This could have been caused by the limited sizes of the sample spaces in the datasets. The conclusion can be made that the designing of complex synthesizers is not in particular a good solution when dealing with small and simple classification problems. Yet the search algorithms could be more beneficial in classification problems with a larger sample set, and a larger number of features and labels. But this theory has not been proven nor denied during this research.

7 Future Work

As mentioned in the conclusion, the performances of the research synthesizers on larger datasets remain untested. This was a goal set for this research project for which the supercomputer of the Technische Universiteit Delft would have been used. But due to complications and time limits, the focus was kept on the smaller datasets mentioned throughout the report. Yet this could be a subject that could be examined and researched further in future work. Once a solution has been found for the runtime irregularities, making it more balanced, the research could continue with larger and more complex datasets. The grammar could be expanded with more transformers and classification algorithms, which would allow a larger search space. Parameters could be tuned more precisely and enumeration and pipeline depths can be increased. The increment of the search space would allow more specific tests that could prove that the machine learning synthesizers have merit compared to the general breadth-first search algorithm on bigger machine learning-related problems.

8 Appendix

```
grammar = Herb.HerbGrammar.@cfgrammar begin
    START   = Pipeline([CLASSIF]) | Pipeline([PRE, CLASSIF])
    PRE     = PREPROC | FSELECT | ("seq", Pipeline([PRE, PRE]))  | ("par", FeatureUnion([PRE, PRE]))
    PREPROC =
        ("StandardScaler", StandardScaler()) |
        ("RobustScaler", RobustScaler()) |
        ("MinMaxScaler", MinMaxScaler()) |
        ("MaxAbsScaler", MaxAbsScaler()) |
        ("PCA", PCA()) |
        ("Binarizer", Binarizer()) |
        ("PolynomialFeatures", PolynomialFeatures())
    FSELECT =
        ("VarianceThreshold", VarianceThreshold()) |
        ("SelectKBest",  SelectKBest(k=4)) |
        ("SelectPercentile",  SelectPercentile()) |
        ("SelectFwe",  SelectFwe()) |
        ("Recursive Feature Elimination",  RFE(LinearSVC()))
    CLASSIF =
        ("DecisionTree", DecisionTreeClassifier()) |
        ("RandomForest", RandomForestClassifier()) |
        ("Gradient Boosting Classifier", GradientBoostingClassifier()) |
        ("LogisticRegression", LogisticRegression())
        ("KNearestNeighbors", KNeighborsClassifier())
end
```

Figure 1: Final grammar.

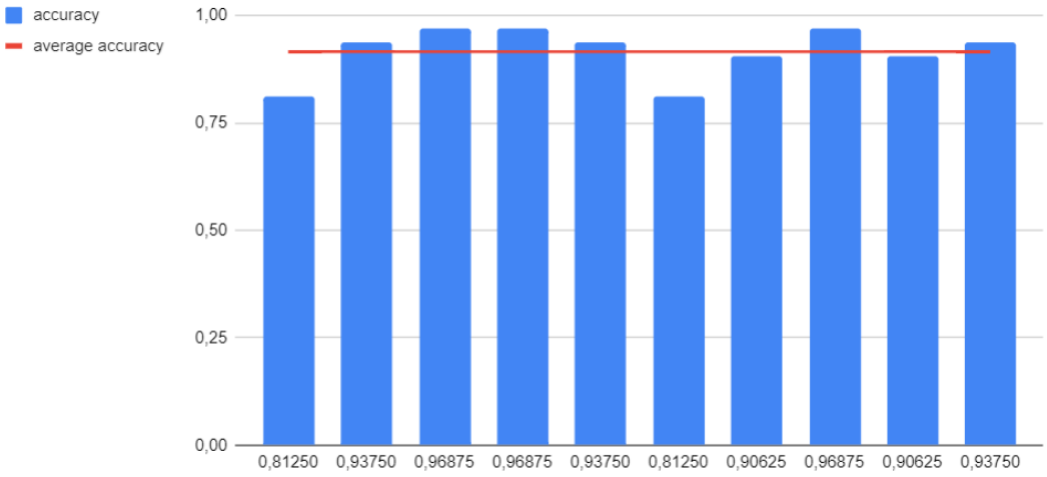

Figure 2: Accuracy graph of enumerative A* search on Seeds dataset.

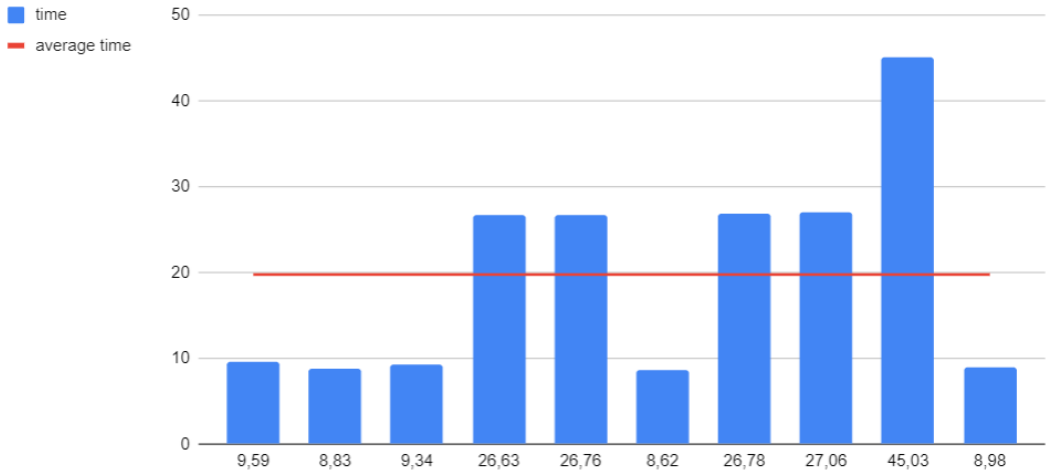

Figure 3: Time graph of enumerative A* search on Seeds dataset.

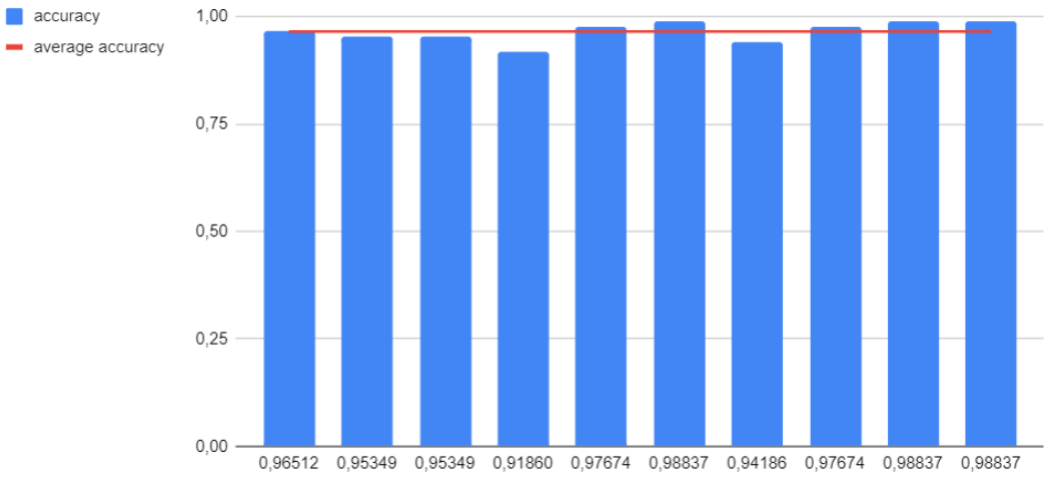

Figure 4: Accuracy graph of enumerative A* search on WDBC dataset.

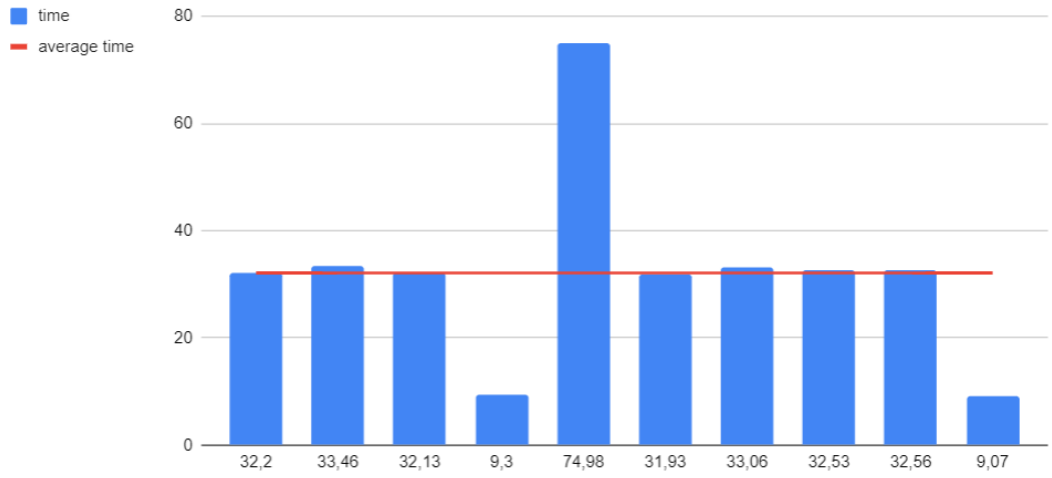

Figure 5: Time graph of enumerative A* search on WDBC dataset.

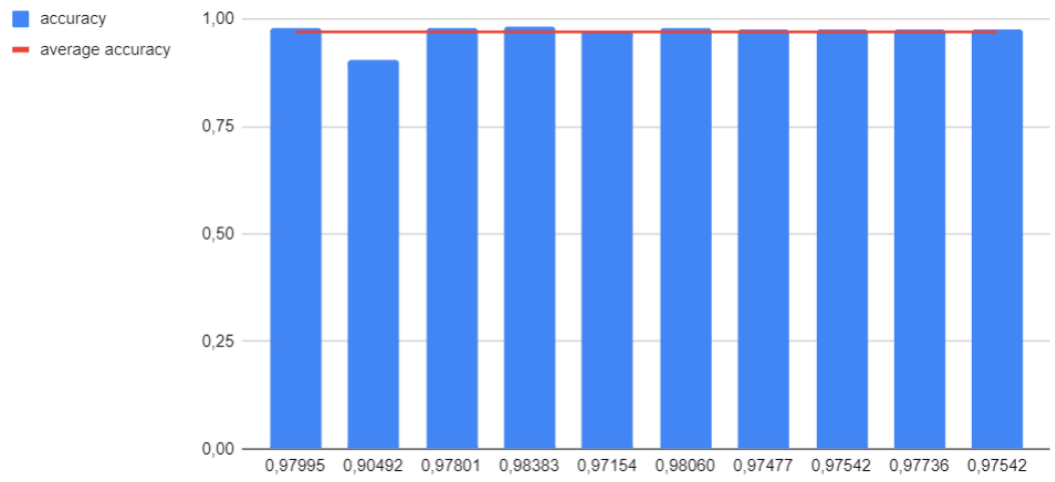

Figure 6: Accuracy graph of enumerative A* search on HAR dataset.

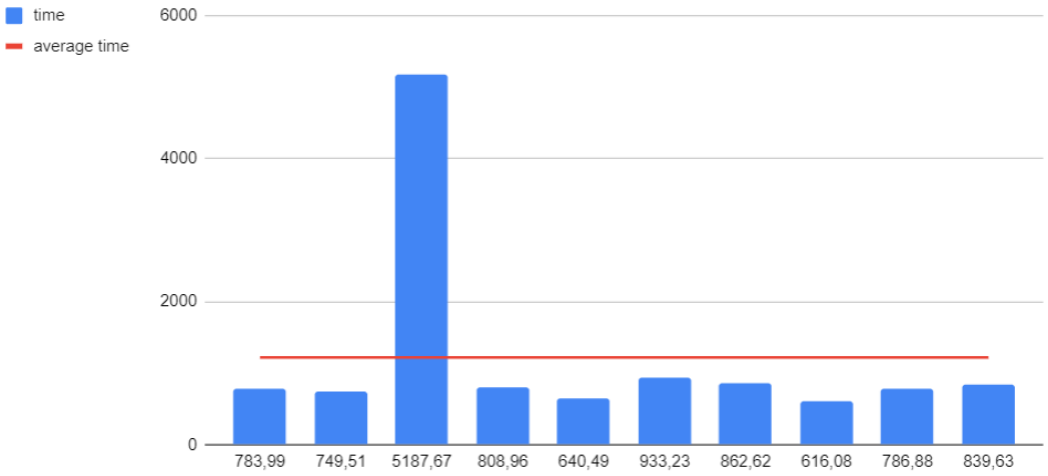

Figure 7: Time graph of enumerative A* search on HAR dataset.

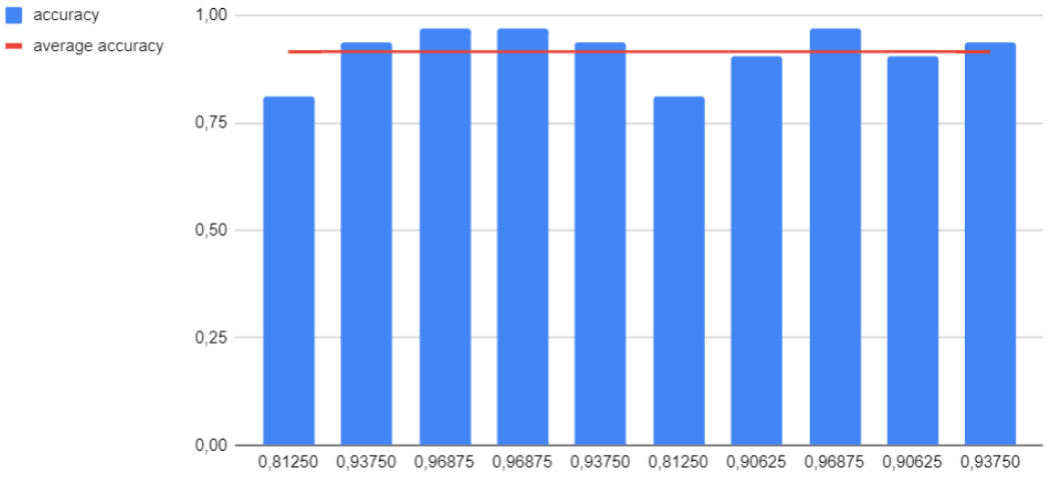

Figure 8: Accuracy graph of breath first search on Seeds dataset.

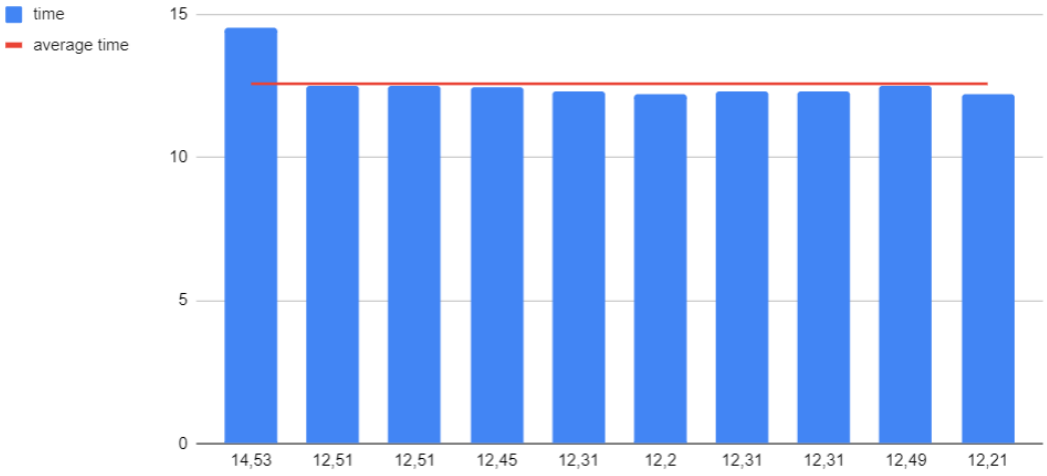

Figure 9: Time graph of breath first search on Seeds dataset.

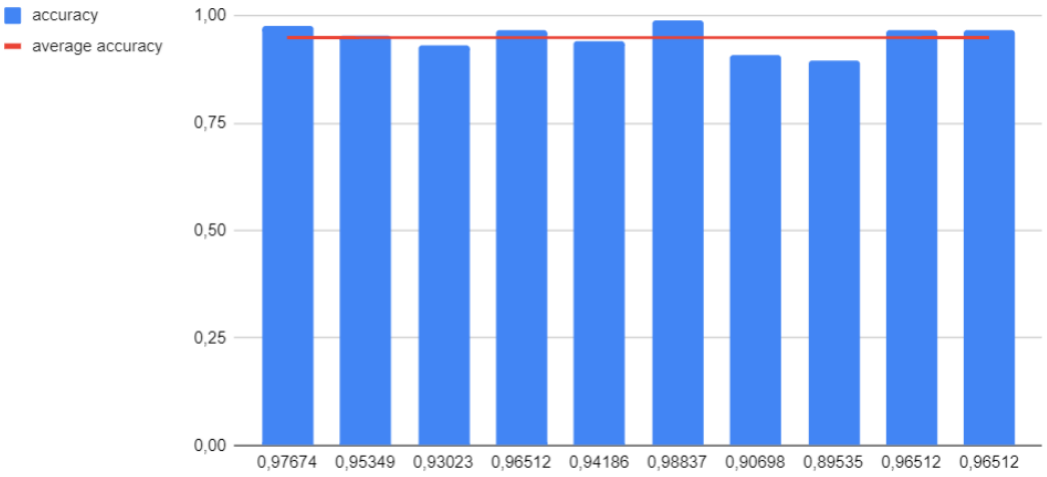

Figure 10: Accuracy graph of breath first search on WDBC dataset.

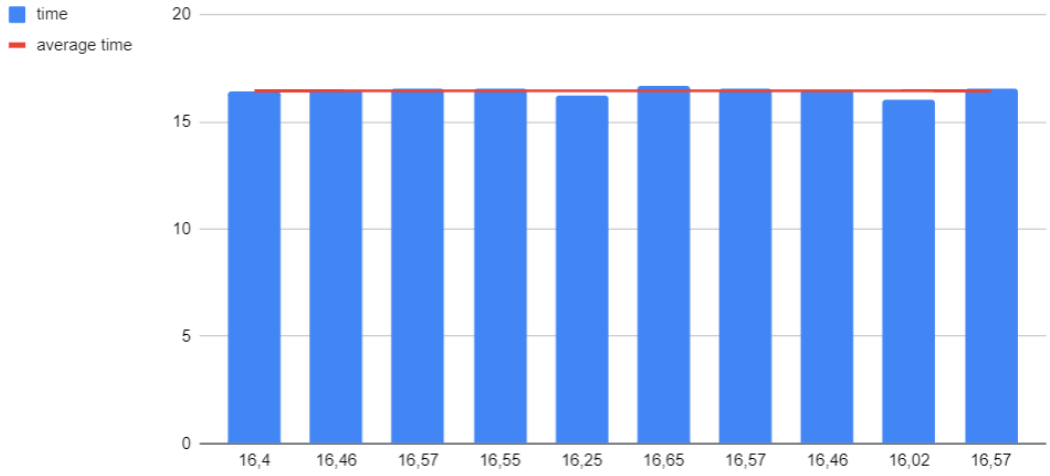

Figure 11: Time graph of breath first search on WDBC dataset.

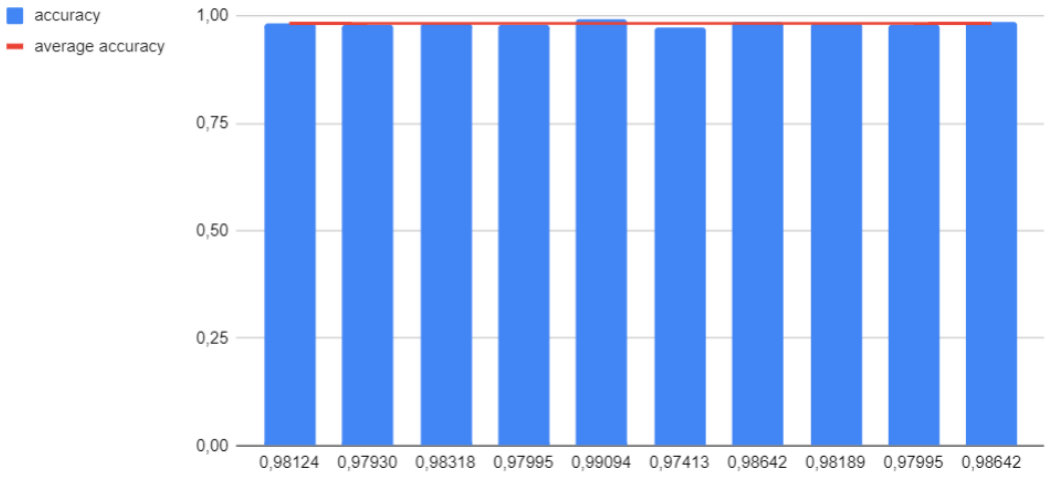

Figure 12: Accuracy graph of breath first search on HAR dataset.

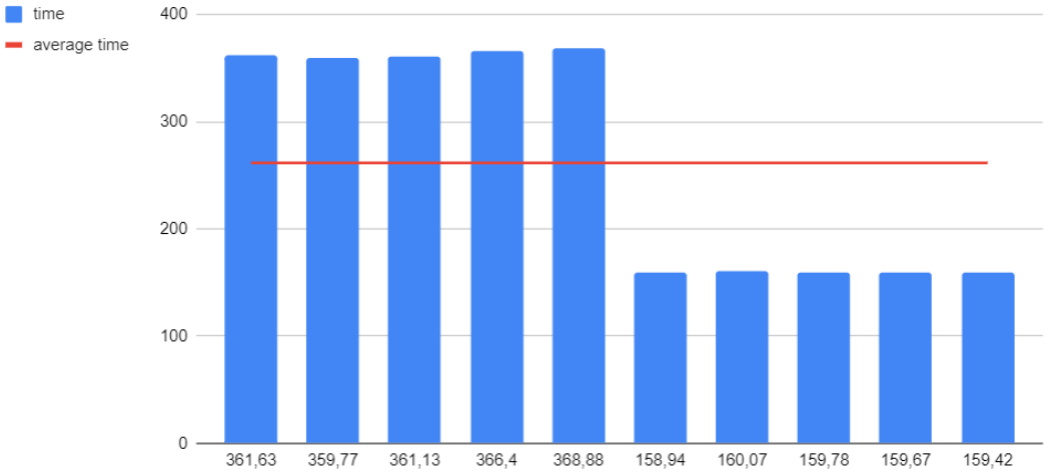

Figure 13: Time graph of breath first search on HAR dataset.

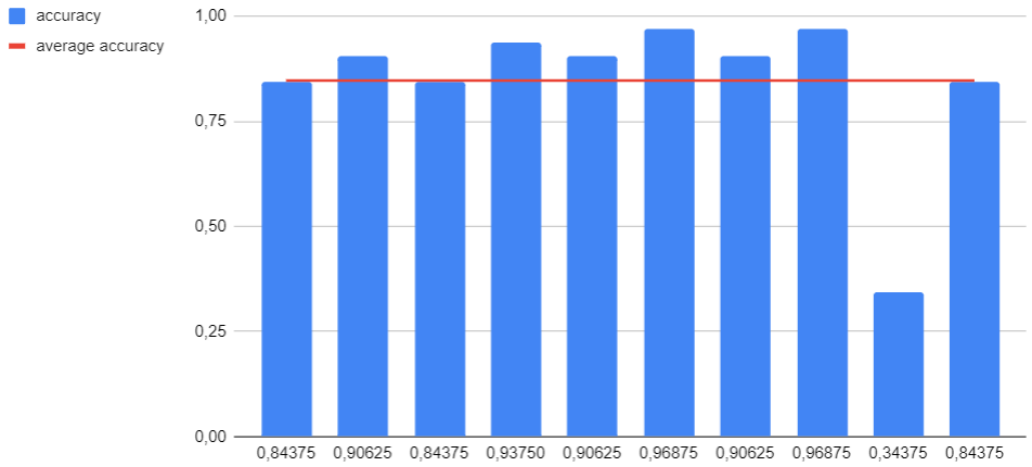

Figure 14: Accuracy graph of the genetic algorithm on Seeds dataset.

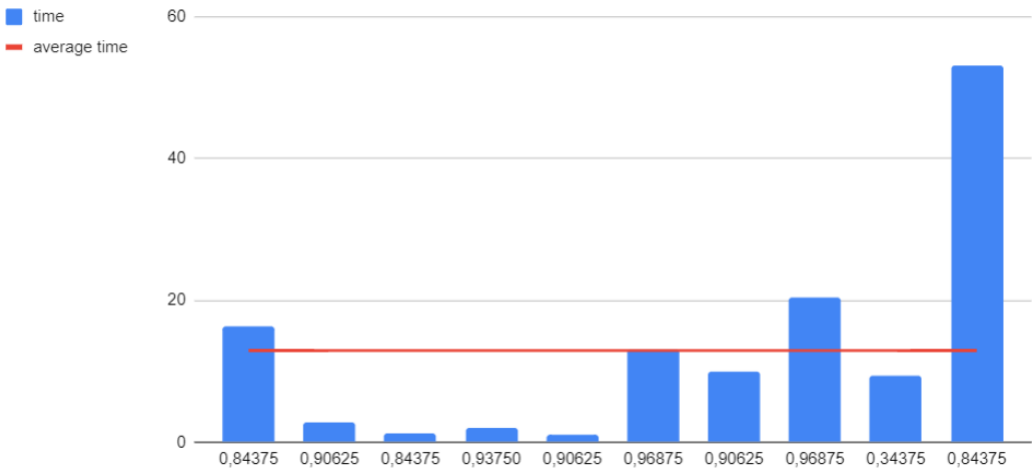

Figure 15: Time graph of the genetic algorithm on Seeds dataset.

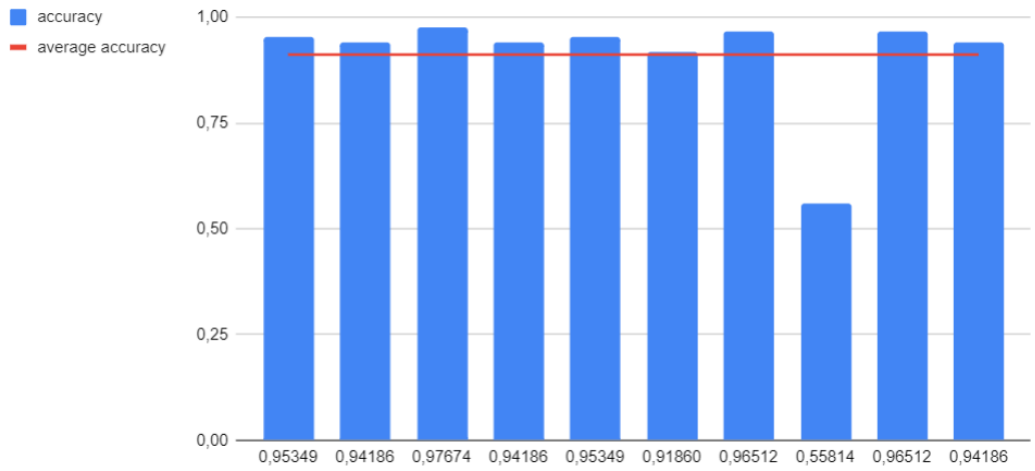

Figure 16: Accuracy graph of the genetic algorithm on WDBC dataset.

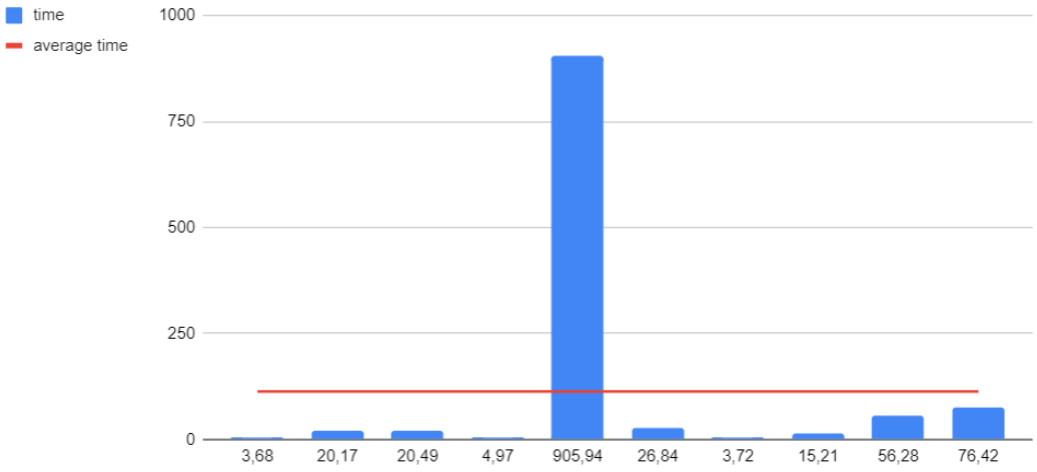

Figure 17: Time graph of the genetic algorithm on WDBC dataset.

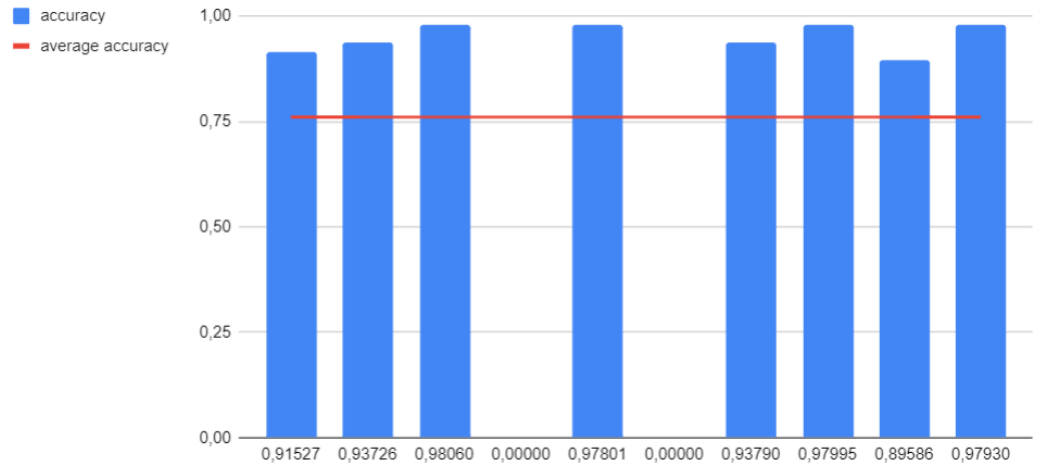

Figure 18: Accuracy graph of the genetic algorithm on HAR dataset.

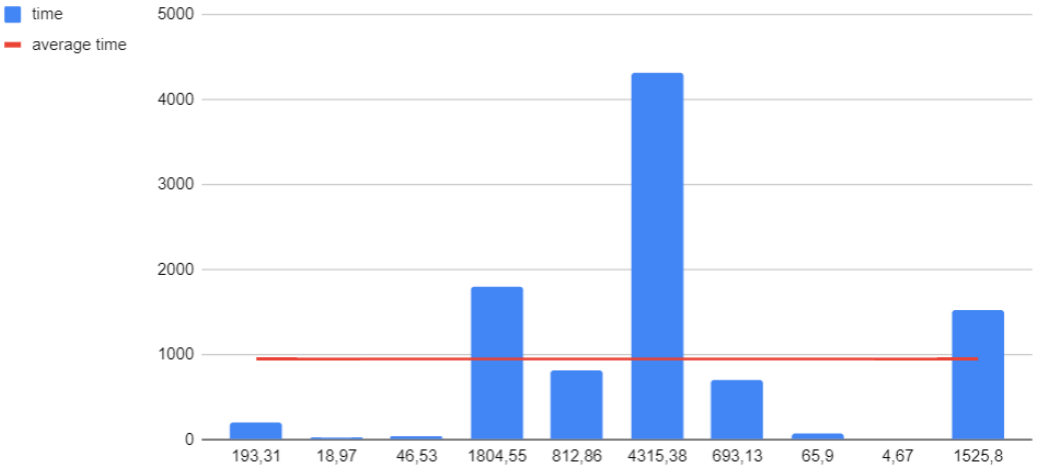

Figure 19: Time graph of the genetic algorithm on HAR dataset.

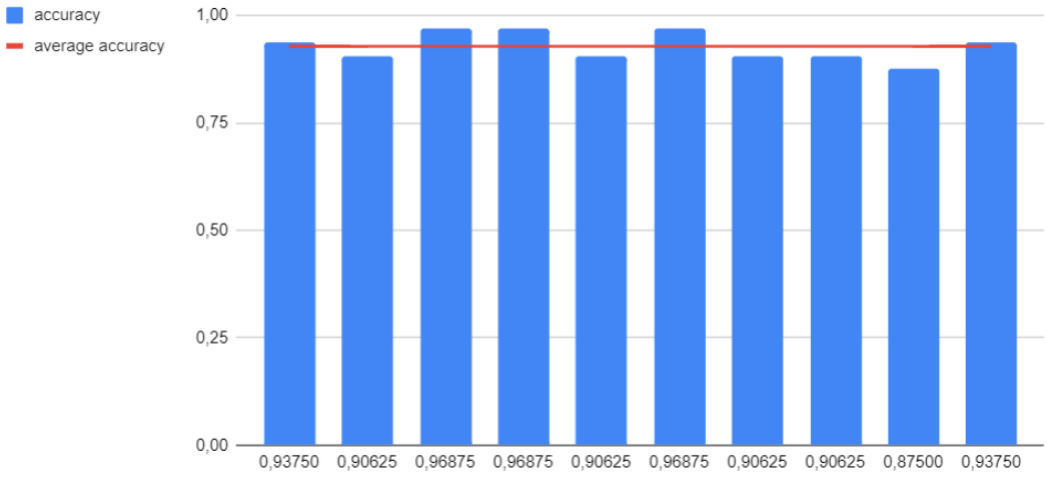

Figure 20: Accuracy graph of Monte Carlo search on Seeds dataset.

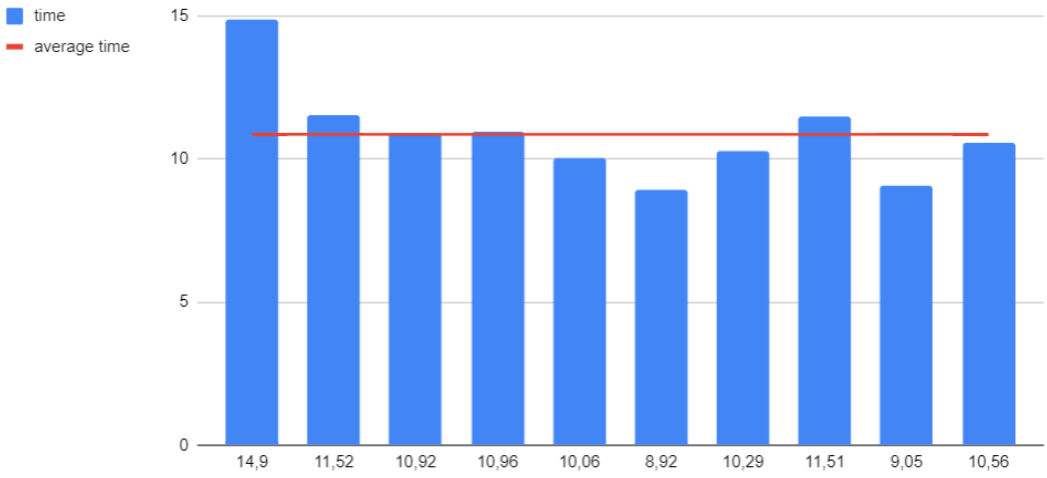

Figure 21: Time graph of Monte Carlo search on Seeds dataset.

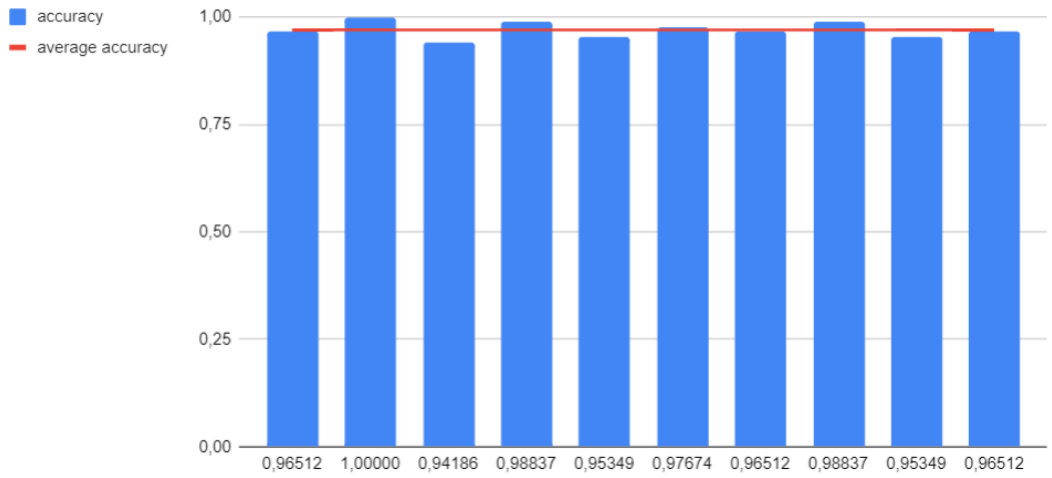

Figure 22: Accuracy graph of Monte Carlo search on WDBC dataset.

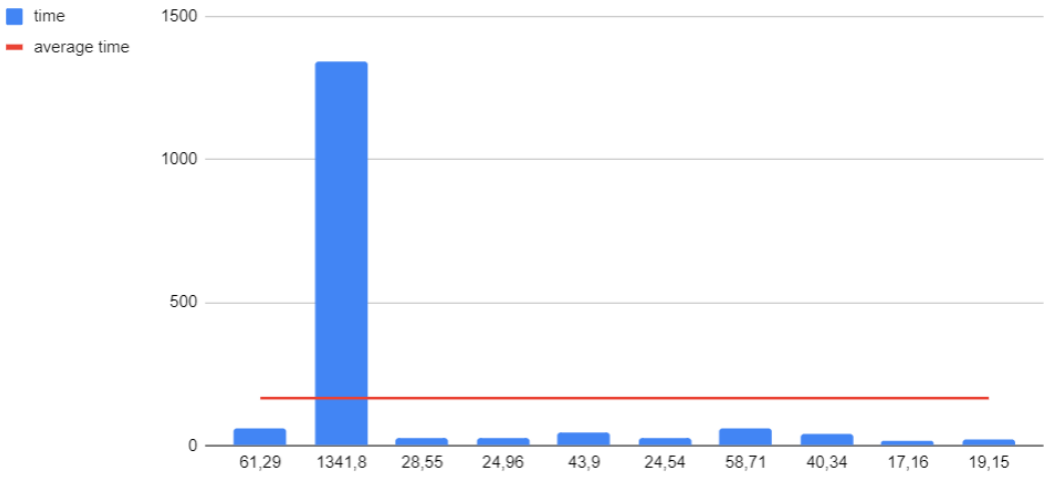

Figure 23: Time graph of Monte Carlo search on WDBC dataset.

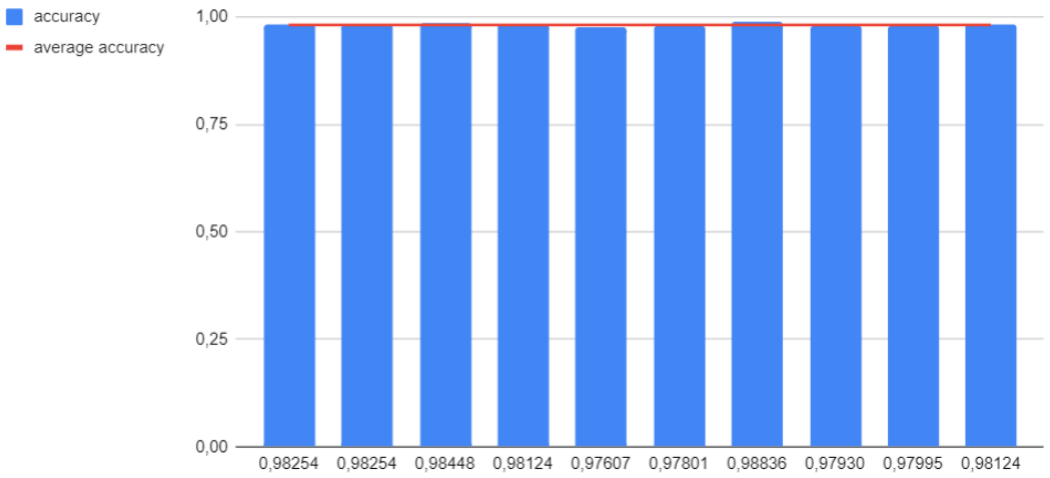

Figure 24: Accuracy graph of Monte Carlo search on HAR dataset.

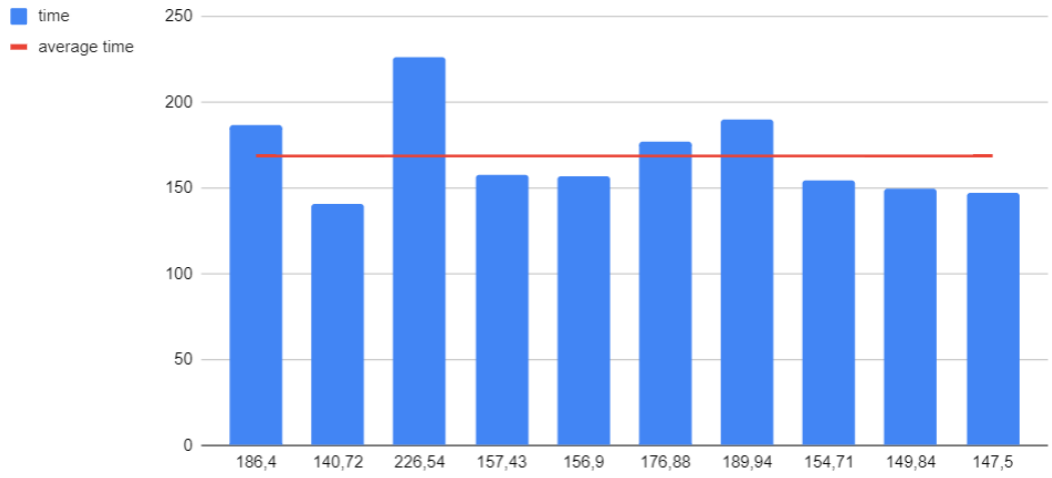

Figure 25: Time graph of Monte Carlo search on HAR dataset.

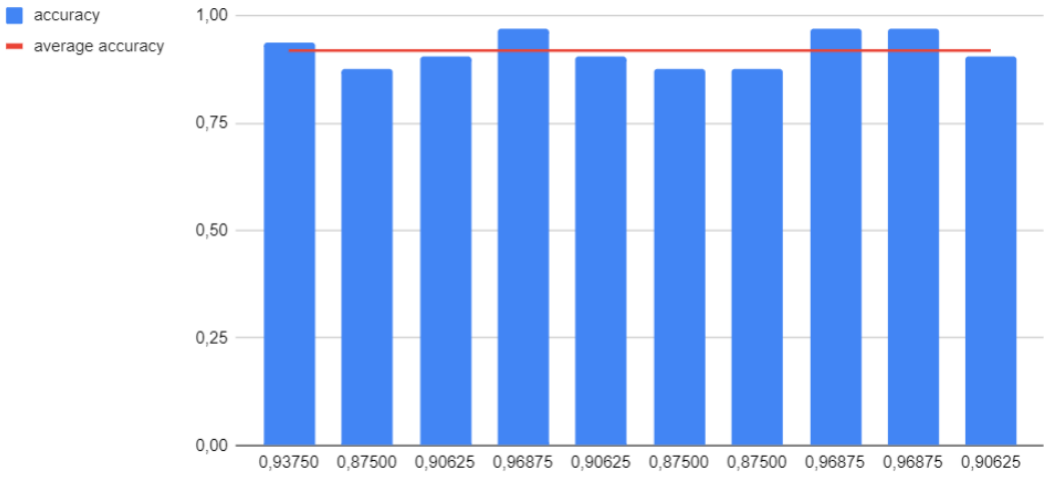

Figure 26: Accuracy graph of metropolis hastings search on Seeds dataset.

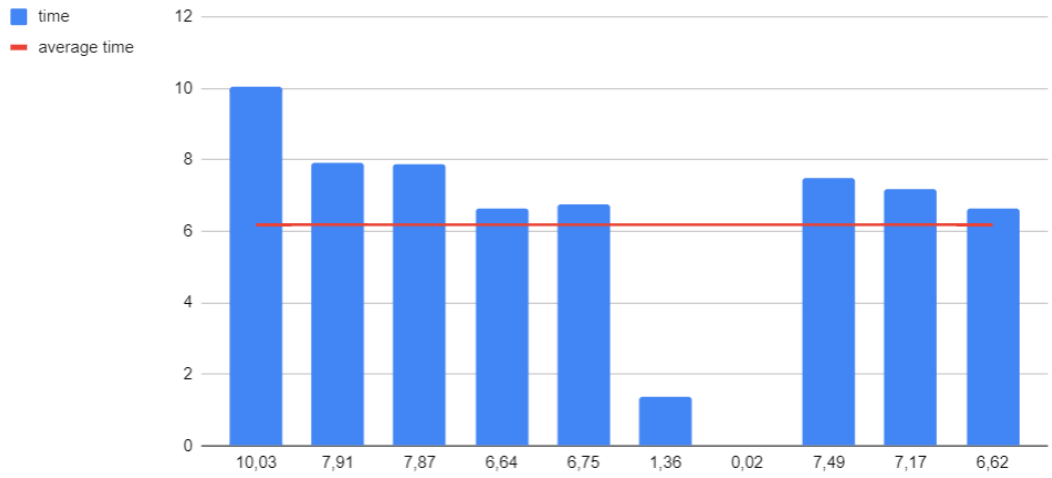

Figure 27: Time graph of metropolis hastings search on Seeds dataset.

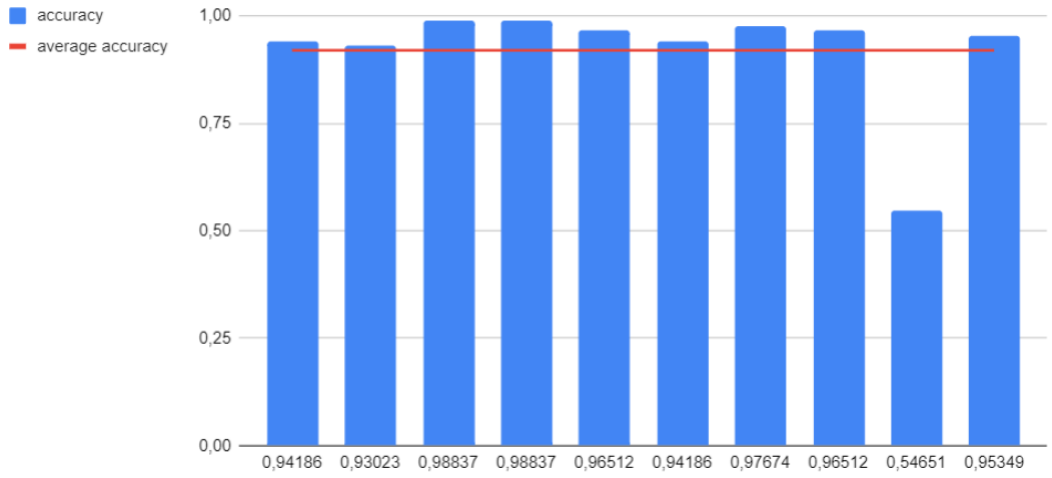

Figure 28: Accuracy graph of metropolis hastings search on WDBC dataset.

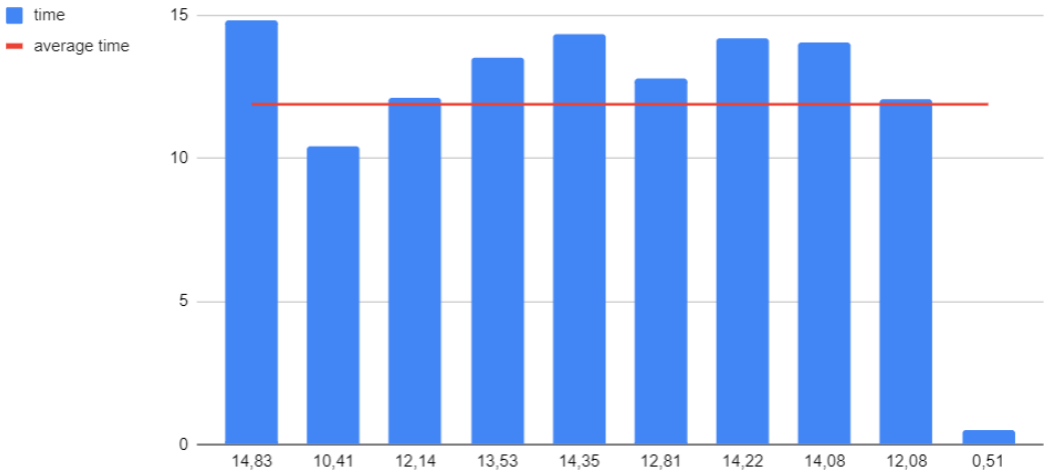

Figure 29: Time graph of metropolis hastings search on WDBC dataset.

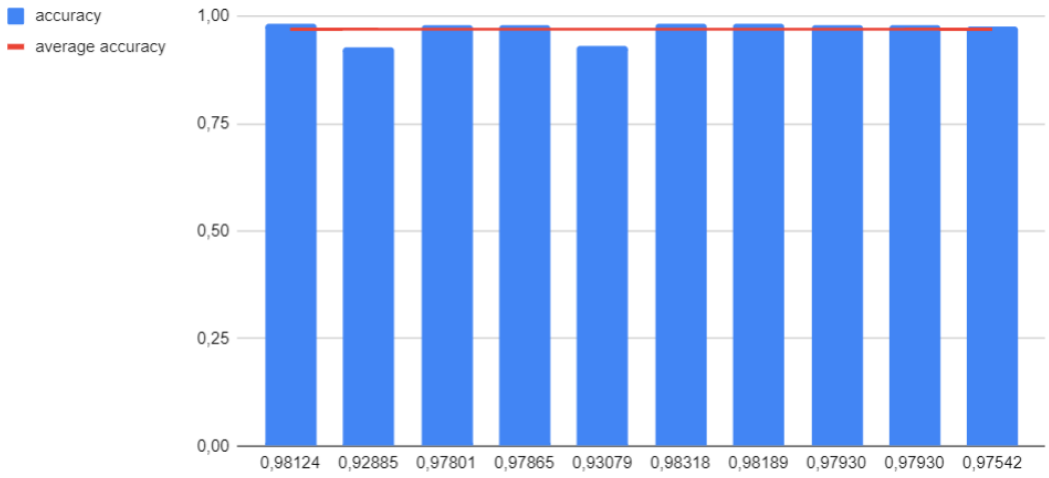

Figure 30: Accuracy graph of metropolis hastings search on HAR dataset.

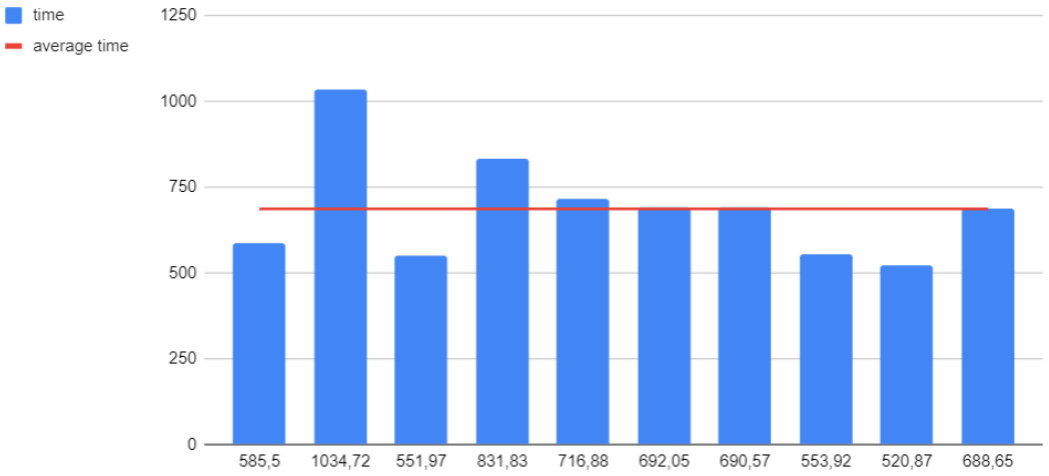

Figure 31: Time graph of metropolis hastings search on HAR dataset.

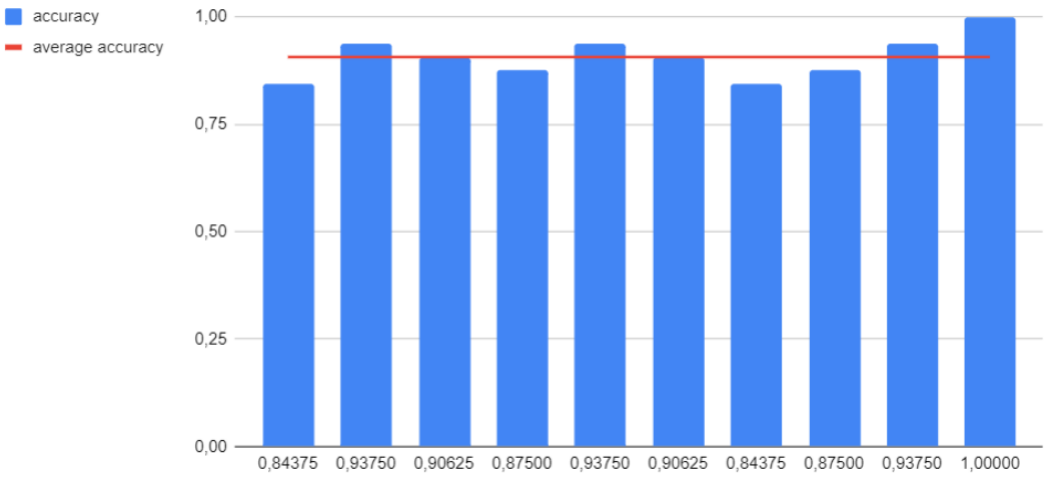

Figure 32: Accuracy graph of very large neighbourhood search on Seeds dataset.

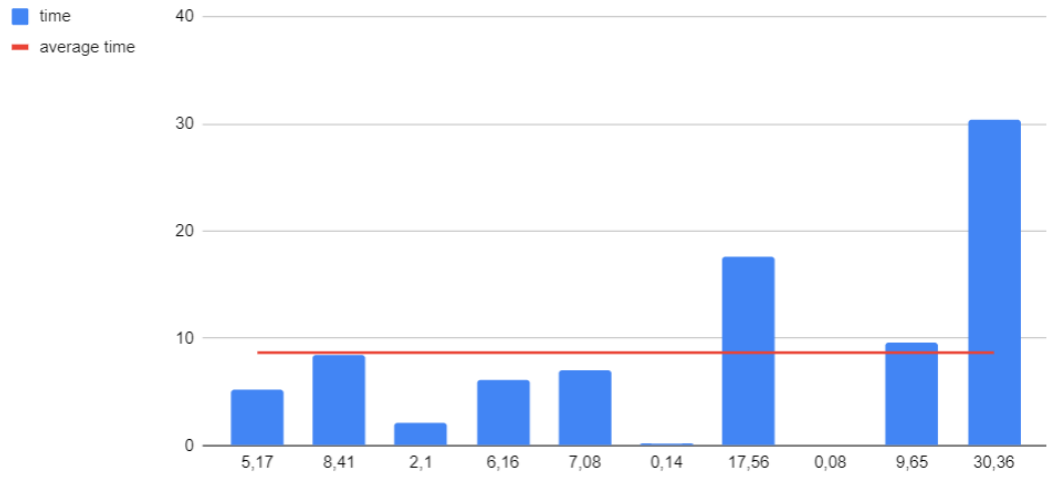

Figure 33: Time graph of very large neighbourhood search on Seeds dataset.

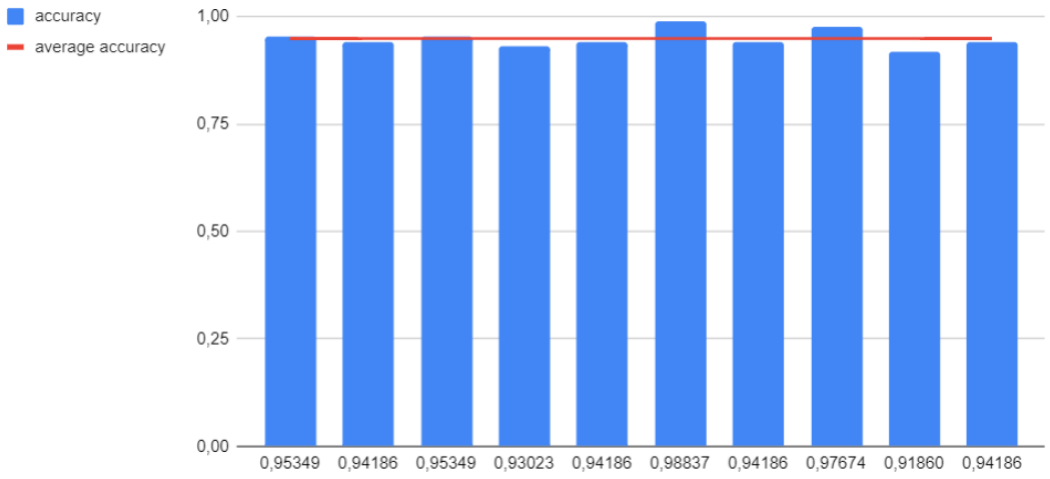

Figure 34: Accuracy graph of very large neighbourhood search on WDBC dataset.

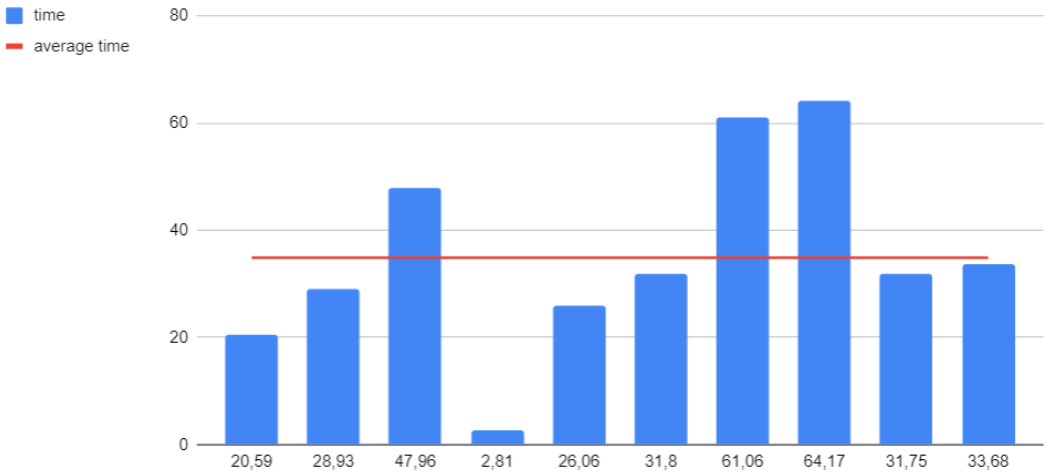

Figure 35: Time graph of very large neighbourhood search on WDBC dataset.

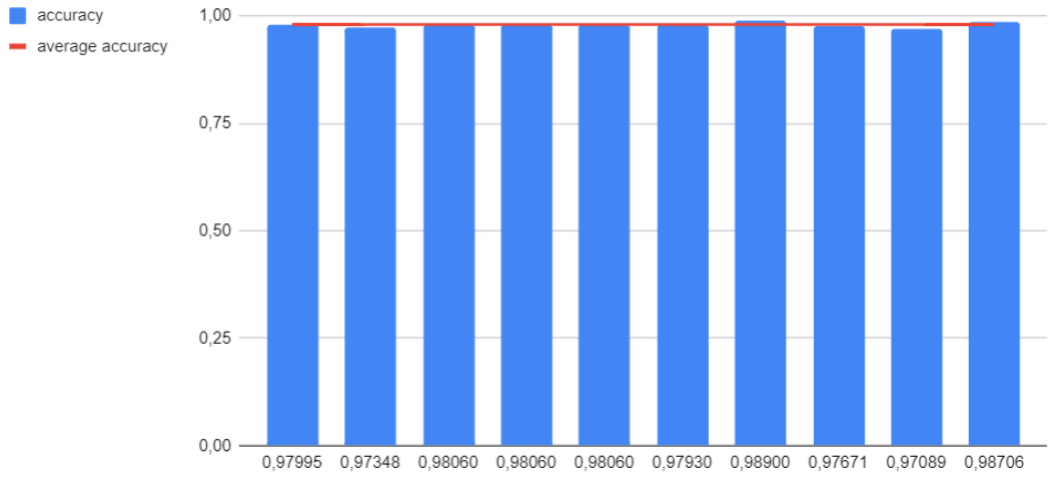

Figure 36: Accuracy graph of very large neighbourhood search on HAR dataset.

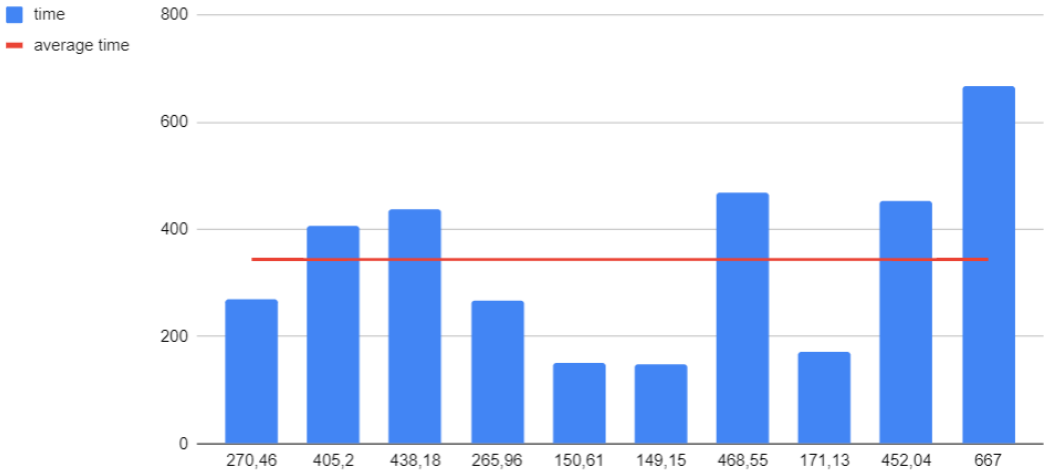

Figure 37: Time graph of very large neighbourhood search on HAR dataset.

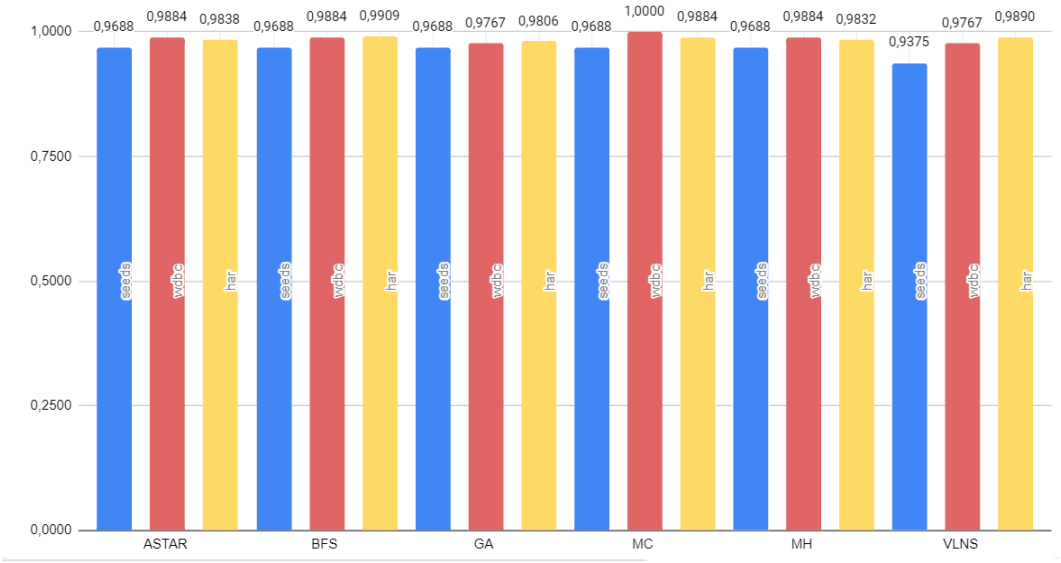

Figure 38: best pipeline accuracy graph of all search algorithms..

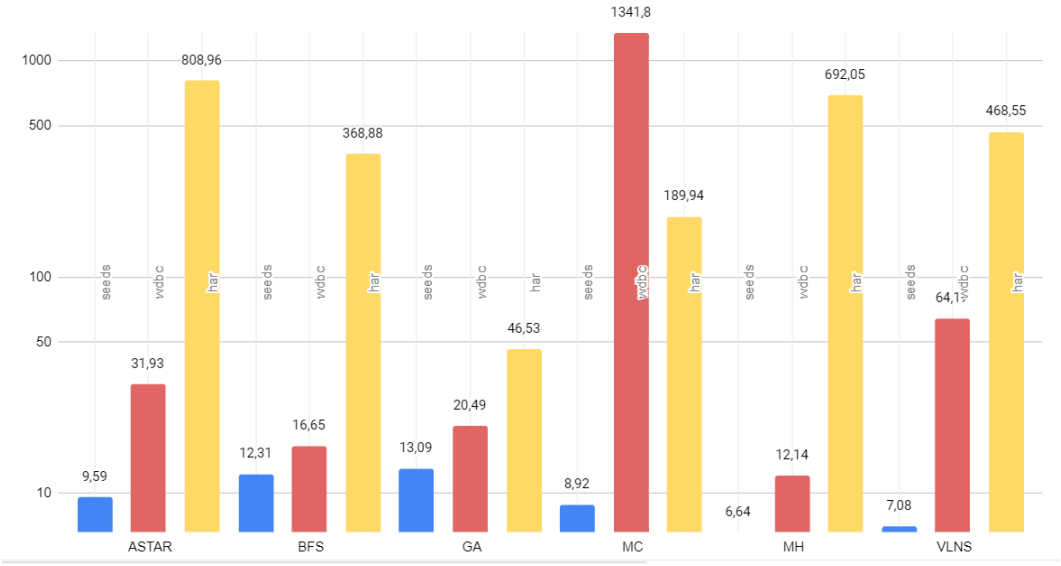

Figure 39: Time graph of all search algorithms.

References

[1] M. Charytanowicz, J. Niewczas, P. Kulczycki, P. Kowalski, and S. Lukasik. seeds. UCI Machine Learning Repository, 2012. DOI: https://doi.org/10.24432/C5H30K.

[2] I. De Falco, âªA. Della Cioppa, and E. Tarantino. Mutation-based genetic algorithm: performance evaluation. *Applied Soft Computing*, 1(4):285–299, 2002.

[3] A. de Sá, W. Pinto, L. Oliveira, and G. 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*, pages 246–261. Springer, 2017.

[4] B. Filius, T. Hinnerichs, and S. Dumanˇ cić. Solving ml with ml: Evaluating the performance of the monte carlo tree search algorithm in the context of program synthesis. *TU Delft preprint: available from repository.tudelft.nl*, 2023.

[5] R. A. Fisher. Iris. UCI Machine Learning Repository, 1988. DOI: https://doi.org/10.24432/C56C76.

[6] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. volume 1 of *Foundations of Genetic Algorithms*, pages 69–93. Elsevier, 1991.

[7] S. Gulwani, O. Polozov, R. Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[8] M. Katz, P. Ram, S. Sohrabi, and O. 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.

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

[10] T. M. Mitchell. *The discipline of machine learning*, volume 9. Carnegie Mellon University, School of Computer Science, Machine Learning âŠ, 2006.

[11] M. Negnevitsky. *Artificial Intelligence: A Guide to Intelligent Systems*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 2001.

[12] J. Reyes-Ortiz, D. Anguita, A. Ghio, L. Oneto, and X. Parra. Human Activity Recognition Using Smartphones. UCI Machine Learning Repository, 2012. DOI: https://doi.org/10.24432/C54S4K.

[13] J. De Jong P. Klop S. Dumanˇ cić, T. Hinnerichs. Herb.jl. `https://github.com/Herb-AI/Herb.jl`, 2023.

[14] D. Sheremet, T. Hinnerichs, and S. Dumanˇ cić. Solving ml with ml: Effectiveness of the metropolis-hastings algorithm for synthesizing machine learning pipelines. *TU Delft preprint: available from repository.tudelft.nl*, 2023.

[15] D. Sheremet, A. Sonneveld, R. Lejeune, B. Filius, M. Butenaerts, S. Dumanˇ cić, and T. Hinnerichs. Herb.jl. `https://github.com/Herb-AI/Herb.jl`, 2023.

[16] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 2006.

[17] A. Sonneveld, T. Hinnerichs, and S. Dumanˇ cić. 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.

[18] A. J. Umbarkar and P. D. Sheth. Crossover operators in genetic algorithms: a review. *ICTACT journal on soft computing*, 6(1), 2015.

[19] W. Wolberg, O. Mangasarian, N. Street, and W. Street. Breast Cancer Wisconsin (Diagnostic). UCI Machine Learning Repository, 1995. DOI: https://doi.org/10.24432/C5DW2B.