Instituto Tecnológico
y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976

Departamento de Electrónica, Sistemas e Informática

Maestría en Diseño Electrónico

TÍTULO DEL TRABAJO.

Machine Learning Algorithms for Small Datasets with Reinforcement Learning and Optimization

TRABAJO RECEPCIONAL que para obtener el **GRADO** de
MAESTRO EN DISEÑO ELECTRÓNICO

Presenta: **Santiago Soule Ascencio**
Director: **Arturo Gómez Cortés**

Tlaquepaque, Jalisco. 14 de Octubre de 2024.

Dedicated to my beloved family, my mother Gabriela and my father Francisco for their support and encouragement.

Acknowledgements

Special thanks to Dr. Omar Humberto Longoria Gandara, and MSc. Arturo Gomez Cortés from the ITESO Department of Electronics, Systems, and Informatics, Mr. Daniel Charles Mendoza from Intel Corporation and MBA. Michael D. Tucknott, for their collaboration, suggestions, and support during the development of this thesis.

I would like to thank the financial support provided by Intel Corporation and all my colleagues who supported me during this journey.

Finally, special thanks to my family for the endless support and encouragement.

Summary

As computers evolve and their computational complexity and performance increase, many forms of machine learning have found new implementations and applications to take advantage of these computer features. Previously, areas dominated exclusively by purpose-built software or fully manual applications have found newly developed innovation from Machine Learning development.

Reinforcement Learning is a way to implement some algorithms that are being explored. Working in a similar way to teaching a young child to perform a task by giving positive or negative feedback depending on their actions, this form of machine learning has great potential with some disadvantages. The results can sometimes greatly outperform computational deterministic implementations.

Being an unexplored area, this investigation has the purpose of implementing and comparing multiple machine learning algorithms with a reinforcement learning optimization algorithm to find out the advantages and disadvantages of each one.

Most of the research involves a reinforcement learning Intel project that would benefit greatly from the results of this investigation and will also suggest that there are certain restrictions that the algorithm implementations must follow. It is also the goal of the investigation to find which algorithm is the most optimal for the Intel project mentioned.

Benchmarking and thorough testing will be done with a real system and implementation of each algorithm to compare objective data during the investigation, as well as an analysis of advantages and harmful properties of each algorithm. Results will be published in the last sections of this document in as much detail as possible without giving confidential information from Intel's intellectual property.

Table of Contents

Acknowledgements v

Summary vii

Introduction 3

Chapter 1: Machine Learning Brief History 6

Chapter 2: Machine Learning Algorithm Landscape 9

2.1 Challenges of Machine Learning 10

2.2 Decision Trees 12

2.3 Random Forests 13

2.4 Extra-Trees 13

2.5 Clustering 14

2.6 K-Means 14

2.7 Neural Networks 14

Chapter 3: Establishing the rules 16

3.1 PMONs 17

3.2 The System Under Test 18

3.3 Evaluation Metrics and Ranking 19

3.4 Graph generation and interpretation 21

Chapter 4: Benchmarking 22

4.1 Decision Tree Regressor 22

4.2 Random Forest Regressor 27

4.3 Extra Trees 33

4.4 KNN 38

4.5 Neural Network 43

Chapter 5: Analysis and Conclusions 45
5.1 Algorithm comparisons 45
5.2 Algorithm Ranking 51
5.3 General Conclusions 53
Appendix 55
A.1 RANDOM FOREST AND EXTRA TREES OPTIMIZER CODE 55
A.2 KNN PREPROCESSOR CODE 58
References 60

Introduction

> [Machine Learning is the] field of study that gives computers the ability to learn without being explicitly programmed.
>
> —Arthur Samuel, 1959

Machine learning is not a new concept and has been around for more than half a century in practical use. Its applications have been very limited until relatively recently with computational power growth and cost reduction that makes it a viable option. Recently, Machine learning, under the guise of 'AI' (artificial intelligence), is making an appearance in many areas previously unheard of, offering higher efficiency and offer better outcomes than traditional software, demonstrating it can offer great potential for many scenarios.

One example of the early implementations is email's spam filters. A very simple algorithm watches what the user usually deletes immediately or even tells the algorithm that an element is unwanted email (a.k.a. spam), and the algorithm creates rules for automatic filtering. Using traditional coding, exhaustive rules would need to be created for pattern recognition in incoming email, but a simple machine learning algorithm is easier to implement and is often capable of demonstrating better performance at this task by resolving the same problems with less effort.

This is an example of the great potential Machine learning has when used for pattern recognition. There are plenty of different algorithms and research already done on this topic, only pushed forward by the great increase in computational power modern technology has enabled in the last 2 decades or so.

Exponential increase in computational power and a significant decrease in cost has enabled many developers, researchers and big organizations to create more applications that use machine learning in many environments, with different complexity levels.

Like any other tools, Machine Learning is not great at everything. There are many tasks for which traditional programming would be a better option, both for ease of development and for better or more reliable results. For example, one of the rules of machine learning (of any type) is that the algorithm may not be 100% accurate, regardless of implementation and method of training. It can get very close, greater than 99% close, but it can never reach total accuracy. For obvious reasons, this is a major drawback in many applications, many of which still use traditional software, such as implementations in which human life depends on its reliability.

There are also certain tasks for which Machine learning is simply either non-applicable, or a terrible choice over traditional software, mainly tasks that require to follow a sequence of actions or ones that require processing of user (human) input since 100% precision would be required and the training cost and complexity would be prohibiting. Another example of this would be that machine learning algorithms follow by nature of being highly optimized for a very particular and specific task and fail in others, which increase use-models complexity.

Machine learning algorithm research and development has advanced greatly with the accessibility to powerful computers. With this also came known algorithms for general implementations, such as Decision Trees, linear regression, vector machines, KNN, etc.

> "A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E."
>
> —Tom Mitchell, 1997

With these advancements, task specialization has become more common. Algorithms were created for specialized tasks such as text and voice recognition, image pattern recognition, classification algorithms and more tools for neural network implementations.

Having a wide variety of algorithms to choose from and implementations available, there is also the discussion of training methods. A machine learning algorithm is only as good as the data it was trained from, if the data was fed to the algorithm properly. There are many ways to train an algorithm, and even more ways to pre-process the data before reaching the algorithm in question.

Because of this, a lot of research has also been done in this subject, often with the goal to optimize the process of training itself (so that it either takes less compute power or needs less data) or optimization for better results and increase accuracy.

One area that has seen very little research is reinforcement learning. Involving the concepts mentioned so far, but by nature a different topic altogether. Yann LeCun famously said that "if intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake".

This thesis presents an analysis of a variety of different machine learning algorithms implemented in an Intel project with the goal of finding the most optimal algorithm for the company's purposes. This will be defined by a series of properties to look out for in the results, as well as a comparison made through benchmarking defined beforehand. All results will be filtered to exclude any company confidential information and otherwise included in this document, with in-depth analysis for each algorithm.

All implementations of Machine Learning algorithms and or the optimizers are made using standard python libraries such as but not limited to ‘SciKitLearn’, ‘Pandas’, ‘Keras’ and ‘numpy’. Testing will be made using a private system with a standard microprocessor and general hardware.

In Chapter 1, a brief history of machine learning is discussed, covering an overview of different notorious examples of machine learning implementations and their evolution, as well as the significant roadblocks that the development faced until now.

Chapter 2 provides an overview of the most important machine learning algorithms, their known advantages, and disadvantages, as well as some more specific details of the ones implemented for the purposes of this investigation, including the main reasons for choosing said algorithms.

Chapter 3 covers the testing methodology as well as environment variable definitions, including description of both set-up conditions as well as the hardware description of the system. A brief explanation for choosing this configuration(s) is also provided.

In Chapter 4, the actual benchmarking and testing is denoted in the most detailed form, providing necessary description of the test results, explanations of said results as well as an in-depth analysis of the properties that the algorithm showed in comparison with the desired objectives of intel’s purposes.

In the General Conclusions, the chosen algorithm according to Intel’s objectives will be described, including the reasoning behind the choice, as well as the characteristics of the algorithms in comparison to the most basic one previously used. The most relevant remarks about this master dissertations are summarized. And finally, appendix A shows the code of certain parts of the implementation that may show how some optimizations were done.

Chapter 1: Machine Learning Brief History

As mentioned in the introduction, machine learning is not. The roots can be traced back to the very first algorithmic formulas developed over half a century ago, but mostly limited due to lack of computational power.

In the following image, a brief recapitulation of some of the most notable events (for machine learning) throughout history is depicted:

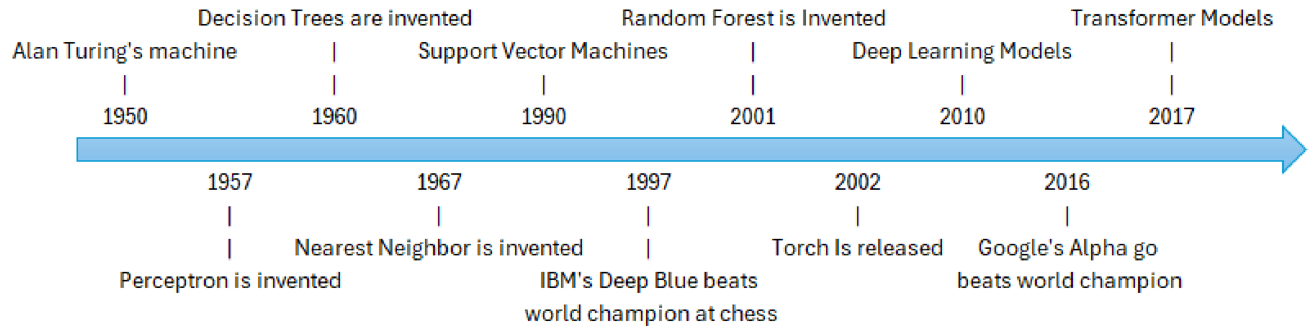

Figure 1: Machine learning timeline

A Perceptron is the precursor of a neuron (in a neural network), and its fundamental version was invented 7 years after Alan Turing attempted to run an algorithm to simulate a person's thinking. While a very remarkable invention, it was first invented as a very particular way of mathematically processing statistical problems.

Most of the machine learning algorithms, in fact, were born this way, algorithms to solve mathematical problems. This is one of the reasons there is a big gap in the timeline where not much happened. It wasn't until 1997 where a supercomputer belonging to IBM, called 'Big Blue' made the headlines when it beat the world champion in chess using an incredibly complicated software coupled with a network of Decision Trees called 'Bayesian network'. (Rogers, 2019)

While Big Blue was a supercomputer by the time, for many people, it was an eye opener in the sense of what AI and Machine Learning could do. Since then, it only took a couple of years for the first open-source library for coding machine learning models, 'Torch' to be released into the public.

Big Blue's achievement was remarkable, but the ML algorithm it used was relatively simple compared to modern ML models. It would take more than a decade for the first deep learning models to appear, and it was until 2016 that one such model made it into news articles again. Google's 'Alpha Go' model, running in another supercomputer, would beat the world champion at the Asian game 'Go'.

Go is a game with rules simpler than those of chess but having a much bigger board and exponentially more mathematically complicated solutions to a particular move. This made it one of the most complicated games that a computer could attempt to play, and Google implemented a deep learning neural network model to do so. This also served as a catapult for other companies to invest more into research with deep learning models and ML in general.

As demonstrated, almost all currently used algorithms were invented before 2010, so the algorithms themselves are (relatively) old, but most advancements and breakthroughs in machine learning implementations are very recent. Henceforth, the real bottleneck is computing power, as demonstrated by figure 2.

'FLOP' or Floating Operations are often used as a unit of measurement in computing systems since it is one of the most taxing operations that a computer can do instruction-wise. To put in context, Deep Blue had around 11.38 GFLOPS (billion flops) and an iphone 5 (released in 2012) had around 76.8 GFLOPS. (Todorov, 2014)

The first implementations of AI and ML needed massive amounts of computing power compared to what was available for general consumers, so it was often limited to government or big corporations in scale that had access to supercomputers. Even if computers have existed before 1950s, it was only after powerful computers became more accessible that ML algorithms really started to develop.

Even with Big Blue's victory in chess in 1997, powerful computers were still very expensive to use, and even then, the computing performance was not enough to implement big models. It wasn't until early 2010s and the release of Torch open-source code libraries for ML that more advanced implementations started to appear.

Just as an example, Google's Alpha Go had around 1.9 million petaFLOP. Comparing that to Deep Blue at the time of the chess tournament, Alpha Go is around 1.6 billion times more powerful. And comparing to a smartphone of the same year as Alpha Go, the iphone7 had 257 GFLOPs according to Apple. This makes the iphone7 135 thousand times more powerful than Big Blue.

As seen in figure 2, a lot more implementations of ML were released near 2020, when now big and powerful computers were much less expensive and more available to the general public, coupled with the now popular cloud computing, using big server computers instead of personal ones.

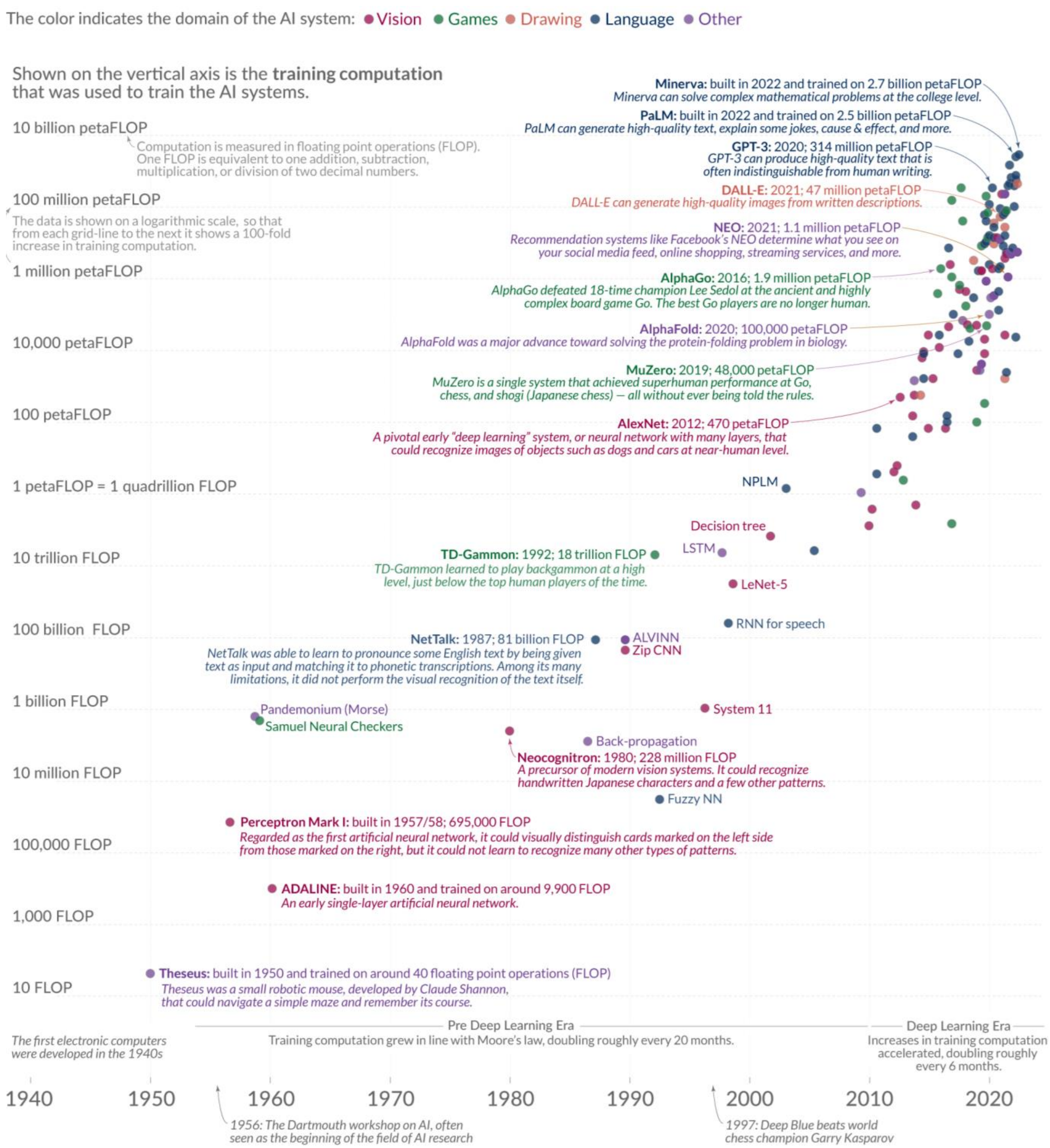

Figure 2: Charted history of exponential growth in AI computation (Our world in Data, 2023)

Chapter 2: Machine Learning Algorithm Landscape

In order to describe the experimentation process, as well as the results of the benchmarking, some concepts need to be defined to avoid inconsistencies and misinterpretation.

Supervised Learning

With supervised learning, an algorithm uses a sample dataset to train itself to make predictions, iteratively adjusting itself to minimize error. These datasets are labeled for context, providing the desired output values to enable a model to give a "correct" answer. Using labeled inputs and outputs, the model can measure its accuracy and learn over time. Supervised learning models are used for two main tasks: classification and regression. (Delua, 2021)

Unsupervised learning

Unsupervised learning algorithms work independently to learn the data's inherent structure without any specific guidance or instruction. Unlabeled input data is provided, and the algorithm tries to identify any naturally occurring patterns in the dataset. Unsupervised learning models are used for three main tasks: clustering, association and dimensionality reduction. (Delua, 2021)

Reinforcement Learning

Reinforcement Learning is very different altogether. The learning system, called an agent in this context, can observe the environment, select and perform actions, and get rewards in return (or penalties in the form of negative rewards). It must then learn by itself what is the best strategy, called a policy, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation. (Géron, Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow, 2019)

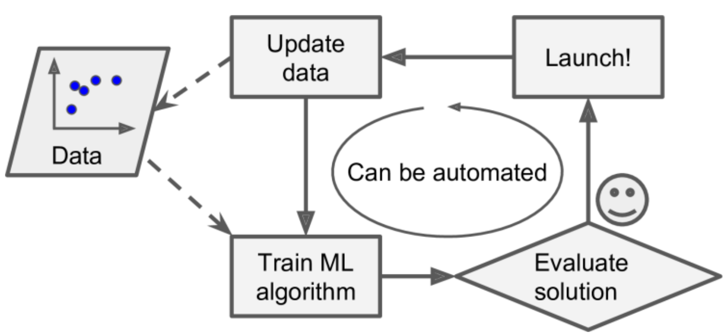

Figure 3. "*Automatically adapting to change*" (Géron, Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow, 2019).

2.1 Challenges of Machine Learning

Since the main task is to select a learning algorithm and train it with data, the two things that can go wrong are "bad algorithm configuration" and "bad data". For the purposes of this thesis, the focus will be on "bad data".

Insufficient Quantity of Training Data

It takes a lot of data for most Machine Learning algorithms to work properly. Even for very simple problems thousands of samples are typically needed, and for more complex problems millions of samples may be needed (unless parts of an existing model can be reused).

Nonrepresentative Training Data

In order to generalize well, it is crucial that training data is representative of the new desired cases. By using a non-representative training set, the model is unlikely to make accurate predictions. If the sample is too small, sampling noise will appear, but even very large samples can be nonrepresentative if the sampling method is flawed. This is called sampling bias.

Poor-Quality Data

If the training data is full of errors, outliers, and noise, it will make it harder for the system to detect patterns. It is often worth the time cleaning up the training data. For example: If some points are clearly outliers, it may help to discard them or try to fix the errors manually.

Irrelevant Features

The system will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones. A critical part of the success is devising a good set of features to train on. This process, called feature engineering, involves:

- Feature selection: selecting only the most useful features to train with among existing ones.
- Feature extraction: combining existing features to produce a more useful one.

(Géron, Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow, 2019)

Overfitting the Training Data

In human terms, Overgeneralization of data. Overfitting happens when the model is too complex relative to the amount and noise of the training data. Mitigations for this include:

- To simplify the model by using fewer parameters or number of attributes in the training data
- To gather more training data
- To reduce the noise in the training data

To avoid overfitting, the model needs to be restricted. The amount of regularization to apply during learning can be controlled by a hyper-parameter. A hyperparameter is a parameter of a learning algorithm (not of the model). As such, it is not affected by the learning algorithm itself; it must be set prior to training and remains constant during training. (Géron, Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow, 2019)

Underfitting the Training Data

Underfitting is the opposite of overfitting: the model is too simple to learn the underlying structure of the data. Mitigations include:

- Selecting a more complex model, with more parameters
- Feeding better features to the learning algorithm
- Reducing the constraints on the model

2.2 Decision Trees

A Decision Tree is a non-parametric supervised learning algorithm, which is utilized for both classification and regression tasks. It has a hierarchical, tree-like structure, which consists of a root node, branches, internal nodes and leaf nodes. (Scikit Learn, 2024)

A Decision Tree starts with a root node, which does not have any incoming branches. The outgoing branches from the root node then feed into the internal nodes, also known as decision nodes. Based on the available features, both node types conduct evaluations to form homogenous subsets, which are denoted by leaf nodes, or terminal nodes. The leaf nodes encapsulate all the possible outcomes within the dataset. (IBM, 2024)

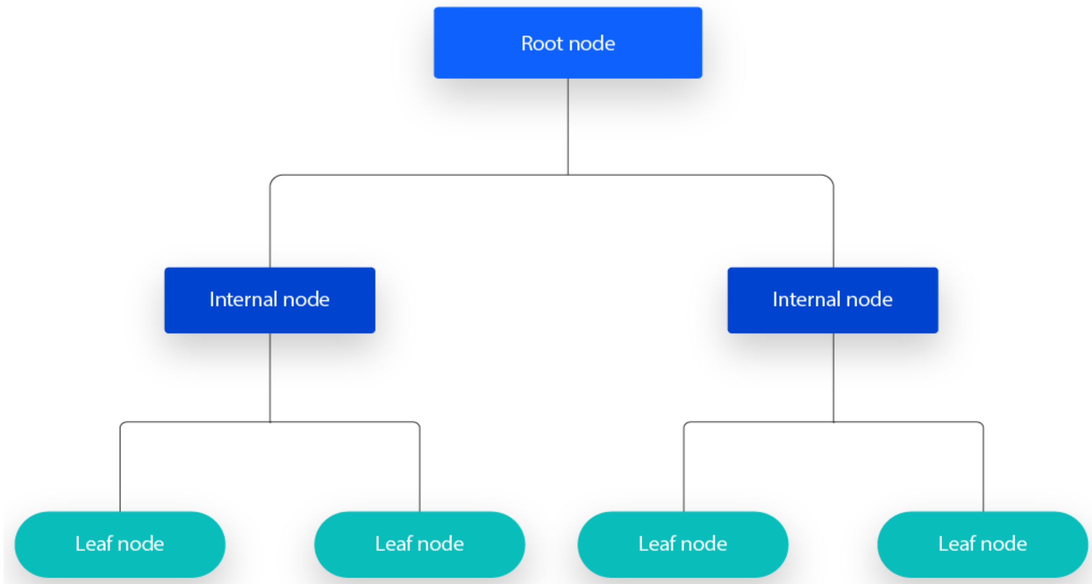

Figure 6. Decision Tree visualization. (IBM, 2024)

Computational Complexity

Making predictions requires traversing the Decision Tree from the root to a leaf. The Decision Tree requires going through roughly O(log2(m)) nodes. However, the training algorithm compares all features on all samples at each node. This results in a training complexity of $O(n \times m \log(m))$. (Géron, Decision Trees, 2019)

Regularization Hyperparameters

If left unconstrained, the tree structure will adapt itself to the training data, most likely overfitting it. Such a model is often called a nonparametric model, because the number of parameters is not determined prior to training, so the model structure is free to stick closely to the data. In contrast, a parametric model such as a linear model has a predetermined number of parameters. (Géron, Decision Trees, 2019)

This is called regularization. For a Decision Tree the most common method is to restrict the maximum depth of the tree. In Scikit-Learn, this is controlled by the ‘max_depth’ hyperparameter.

2.3 Random Forests

If the predictions of a group of predictors are aggregated, better predictions will often be achieved compared with the best individual predictor. A group of predictors is called an ensemble; thus, this technique is called Ensemble Learning.

The sklearn.ensemble module includes two averaging algorithms based on randomized Decision Trees. These algorithms are perturb-and-combine techniques specifically designed for trees. The prediction of the ensemble is given as the averaged (rounded) prediction of the individual classifiers. (Scikit Learn, 2024)

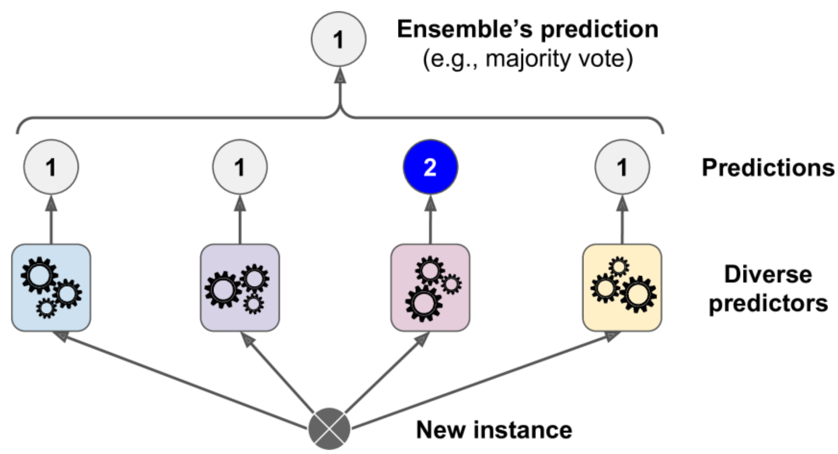

Figure 7. Random Forest Classifier composition diagram. (Géron, Decision Trees, 2019)

2.4 Extra-Trees

When growing a tree in a Random Forest, at each node only a random subset of the features is considered for splitting (bifurcation of 2 or more tree nodes). It is possible to make trees even more random by also using random thresholds (decision-making parameters) for each feature rather than searching for the best possible thresholds. A forest of such extremely random trees is simply called an Extremely Randomized Trees ensemble (or Extra-Trees for short). (Géron, Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow, 2019)

2.5 Clustering

An expert may be needed to tell what type a particular object is, but it is certainly not needed to identify groups of similar looking objects. This is called clustering: it is the task of identifying similar instances and assigning them to clusters.

Clustering is used in a wide variety of applications, including (but not limited to):

- As a dimensionality reduction technique
- For anomaly detection (also called outlier detection)
- For semi-supervised learning

2.6 K-Means

The K means clustering algorithm assigns data points to one of the K clusters depending on their distance from the center of the clusters. It starts by randomly assigning the clusters centroid in space. Then each data point is assigned to one of the clusters based on its distance from centroid of the cluster. After assigning each point to one of the clusters, new cluster centroids are assigned. This process runs iteratively until it finds an optimal distribution. The number of clusters is given in advance. (Scikit Learn, 2024)

K Means clustering performs best when data is well separated. When data points are overlapped, this clustering method is prone to significant errors. K Means is faster compared to other clustering techniques as it provides strong coupling between the data points.

2.7 Neural Networks

A neural network is a machine learning program, or model, that makes decisions in a manner similar to a real brain, by using processes that mimic the way biological neurons work together to identify phenomena, weigh options and arrive at conclusions.

Every neural network consists of layers of nodes, or artificial neurons—an input layer, one or more hidden layers, and an output layer. Each node connects to others and has its own associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network. (IBM, 2024)

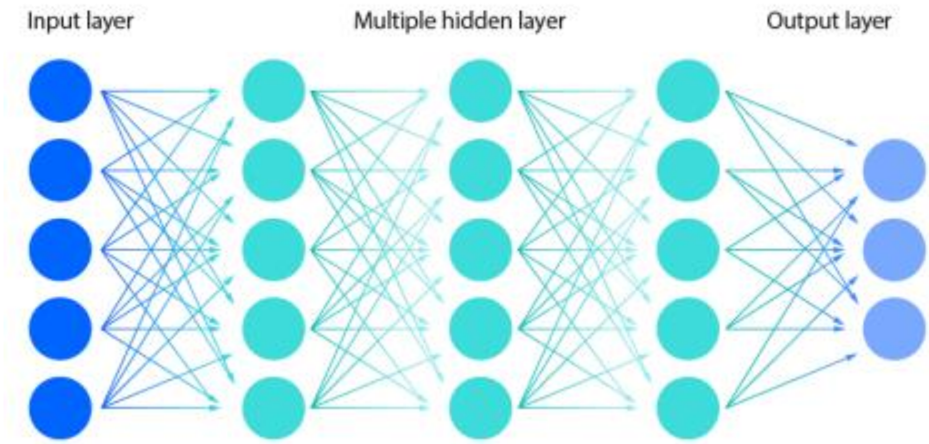

Figure 8. Example diagram of a neural network. (IBM, 2024)

Neural networks rely on training data to learn and improve their accuracy over time. Once they are fine-tuned for accuracy, they are powerful tools in computer science and artificial intelligence, allowing us to classify and cluster data at a high velocity.

That being said, Neural networks in general are (to humans) the most complicated machine learning implementation to understand, especially as the network grows in scale depending on the implementation. This makes it harder to manually tune and also adds a certain uncertainty since the outputs can vary from each training of algorithm (NN are naturally non-deterministic algorithms). (Amazon Web Services, 2024)

A neural network that consists of more than three layers—which would be inclusive of the inputs and the output—can be considered a deep learning algorithm. A neural network that only has two or three layers is just a basic neural network. (IBM, 2024)

Chapter 3: Establishing the rules

For this application, we are using a simple Decision Tree regressor as the original algorithm supported. Due to the nature of the framework and environment in which the application runs, Decision Tree is probably the simplest and most straightforward algorithm to use. However, being the simplest does not mean it is necessarily the best in all aspects, nor the most optimal.

According to Cambridge Dictionary, benchmark is defined as: “to measure the quality of something by comparing it with something else of an accepted standard”. To define what ‘an accepted standard’ is, a set of environment variables and configuration will need to be defined, such as the platform to use, what type of test to run, means of measurement, etc.

For the purposes of this thesis, the original implementation of a Decision Tree regressor will be considered as the reference with as little modifications as possible to establish a baseline to beat (or benchmark) with the other algorithms developed for this thesis.

The goal is to compare four algorithms with initial conditions and environment as little variation as possible, effectively benchmarking all algorithms against the original one. To achieve this, a set of rules were defined:

1. The measurement setup shall be the same
2. The test goals between algorithms shall be the same
3. The test(s) duration shall be the same between runs
4. The platform in which the tests run shall be the same for all tests
5. The platform configuration shall remain the same between tests
6. Starting conditions for each test shall remain the same
7. All external factors shall be minimized as much as possible and annotated when they cannot be removed and affect observations.

For point 1, PMONs will be used. PMONs or ‘Performance Monitor Counters’ are a feature of Intel architecture that allow to count a variety of different transactions running in the system in real-time. (Intel, 2024)

3.1 PMONs

For these tests, a set of 3 different counters will be used with different types of transactions. The counters were chosen with transaction types intended to not be uncommon but also not the easiest to maximize:

- INST_RETIRED (retirement of the last Uop of the instruction)
- UNC_CHA_LLC_VICTIMS (LLC stands for ‘Last Level Cache’, lines that were victimized)
- Internal PMON 1 (this should be similar in difficulty and or complexity to LLC_victims)

A synthetic test aimed toward core-cache transactions will be used, with run parameters. The software version of this test generator will remain the same for all tests.

The objective for all algorithms will be the maximization of the above counters normalized against the highest value of each experiment, using the traffic generator parameters to create the appropriate traffic for each counter.

‘INST_RETIRED’ counter was chosen for being the ‘easy case’, meaning that any algorithm should be able to increase the value without many complications. ‘UNC_CHA_LLC_VICTIMS’ was chosen as the ‘hard case’; while not considered extreme, it should pose a considerable challenge to the algorithms. And ‘Internal PMON 1’ should be considered as a middle ground.

3.2 The System Under Test

The traffic generator is designed to run in 'fragments' (which will be referred to as such during the rest of this document) during which the specified type of traffic selected through parameters is executed while measured. The decisions shown by the machine learning model will be the feedback of the reinforcement learning. The Machine learning algorithm should then use the newly acquired information to constrain some of the parameters (configuration) used in the next experiment for the test generator to maximize the chosen counter. Each counter will be tested independently for each algorithm and will be analyzed accordingly.

The SUT (system under test) will be an Intel processor with 16 cores and 16 GB of RAM, no hyper-threading, virtualization or clustering enabled. No PCIe, CXL or Optane devices will be used. The platform is running Linux with only the most basic drivers loaded, and nothing but the traffic generator is running during the tests.

It is also worth noting that both the OS used and the software for the tests use internal Intel-developed libraries that enable access to low-level hardware features, such as direct register access and physical memory address configurations.

To minimize variability and interference, counters are assigned to a specific core in the system, in this case being core 4, and all basic OS operations will be assigned an affinity to core 0. The Linux operating system may schedule other operations on these cores and that would be annotated when observed. There will also be a 'host' system (a windows computer with specialized software) connected with J-TAG to the SUT. The host executes python and the actual Machine Learning algorithm, passing the commands for the test generator to the SUT.

No OS or BIOS configuration will be changed between tests. All previous test files will be moved to the host system and then deleted from the execution platform to avoid any type of conflict between tests. On the host system, all tests will be started with a pre-defined starting point and 'starting dataset' as described below.

All algorithms are provided with a pre-generated dataset of 100 fragments to help accelerate testing times. All algorithms are provided with the same initial dataset. Said dataset was generated with the same SUT, left to run without any constraints. The configuration, parameters and 'history' of the training during the generation of the initial dataset were erased afterwards, to ensure a clean state for the actual testing. The test duration will be capped at 1000 fragments of data unless there is a special characteristic observed that may prove useful in the analysis.

All algorithms are mostly composed of 3 parts: the pre-processor (responsible for parsing data so that the ML model can read it), the ML implementation (the model itself), and an optimizer, that iteratively regulates and adjusts some hyper-parameters (based on model scoring (Scikit Learn, 2024)). Aside from these parts, everything else in the test remains the same (both software and hardware), meaning that any change in the results is only caused by the change of algorithms and the counter chosen to maximize.

3.3 Evaluation Metrics and Ranking

Having determined the testing methodology, the evaluation criteria is also needed to evaluate which algorithm is better suited for a determined task. For the intentions of this investigation, there are 4 main factors to evaluate:

1. Speed of convergence: defined as the number of samples needed to reach a 'stable' output
2. Maximum average value reached: defined as the output value once the stability is reached
3. Variance during stability period: defined as the variation/spread of output values once stability is reached
4. Absolute maximum value reached: defined as the maximum output value reached during the entire test

The 'stability' talked about above refers to the state the output reaches when the values visually 'settle' or converge to a (visual) line in which the output tends to stay within a relatively small margin of a certain value. For this document, stability will be determined by the 'contour' of the graph, since there will always be samples below margin due to a percentage of randomization.

Since reinforcement learning in this case also involves an executable spanning multiple seconds in the SUT for each datapoint generated (or iteration), coupled with trial-and-error and hyperparameter tuning, it takes an extremely long time to reach satisfactory results, considering thousands of iterations may be required. With this in consideration, Speed of convergence takes priority above the others, but may not be the only criteria since there are times when the others may be prioritized.

Considering the evaluation criteria, a ranking system will also be implemented, using the weight of an evaluation point (ranging from 1 to 4), and which place each algorithm performed in contrast to each other. These two numbers will be multiplied and then added for all tests (3 counters). The algorithm that has the highest value will be considered 'the best option' overall.

Algorithm score for counter 1 = [metric 1 weight] / [algorithm performance place (1-4)] + [metric 2 weight] / [algorithm performance place (1-4)] + [metric 3 weight] / [algorithm performance place (1-4)] + [metric 4 weight] / [algorithm performance place (1-4)]

Algorithm overall score = score for counter 1 + score for counter 2 + score for counter 3

Weights for scoring metrics:

- Speed of convergence = 4
- Maximum average value reached = 3
- Divergence during stability period = 2
- Absolute maximum value reached = 1

There may be scenarios in which another algorithm would be more desirable. This will be covered in more detail, along with a detailed analysis in ***chapter 5*****.**

3.4 Graph generation and interpretation

Average graphs (figure 10 as an example) were created by calculating the average of every 10 samples (data points 1-10 make the first data point in the average graph, 11-20 make the second and so on). Similarly, maximum graphs (figure 11 as an example) were created by taking the maximum value of every 10 data points.

While all scatter plot graphs (figure 9 as an example) are normalized against the highest value of that same graph (so that Y-axis remains between 0 and 1), maximum and average graphs are also normalized against their own highest value. For example, after the process of obtaining average data points described above, this new set of data points is normalized, so that the Y-axis remains between 0 and 1, regardless of the type of graph.

For overlay graphs (figures 48 and onwards) a slight change in the process was made. First, the average values are obtained for each algorithm in their corresponding counters. Then, the corresponding data for each algorithm is overlayed, and after this the entire overlay is normalized. The intention is to constrain the Y-axis between 0 and 1, but still show the scaling and contrast in behavior between the algorithms.

For overlay graphs, some of them may be slightly truncated due to difference in sample size for certain algorithms. The X-axis will be truncated to the smallest sample size for all algorithms. For example, if 3 algorithms have a sample size of 100 but one has a sample size of 60, the overlay graph would have a sample size of 60.

Normalization of graphs was done with the purpose of making it easier for the reader to visualize the scales in simple numbers, and to avoid exposing Intel's proprietary information with reverse engineering.

Chapter 4: Benchmarking

All training is executed in an iterative manner, using positive or negative reinforcement learning. The algorithm should analyze the current dataset and generate an output for the SUT to execute, generating PMON numbers in return. While explicitly assigning the algorithm the task to maximize a specific counter, the algorithm as a whole can take the value of the counter itself as feedback.

All algorithms include a 5 to 10 percent randomization in all iterations to help 'stagnation' (in which the algorithm finds a local maximum but refuses to search for a higher value).

NOTE: for direct comparisons between algorithms, please refer to chapter 5.

4.1 Decision Tree Regressor

Being the most basic implementation, there is virtually no pre-processor as the data is passed on directly to the algorithm with only the minimum alteration so that the algorithm can understand it (in other words, a parser) and the actual ML implementation is a Decision Tree Regressor from Sci-kit Learn library.

The optimizer in this case makes an analysis of all the tree's leaf values and parameters, then takes the group of parameters and values with the highest outcome of the counter and uses that as the outcome of the entire algorithm.

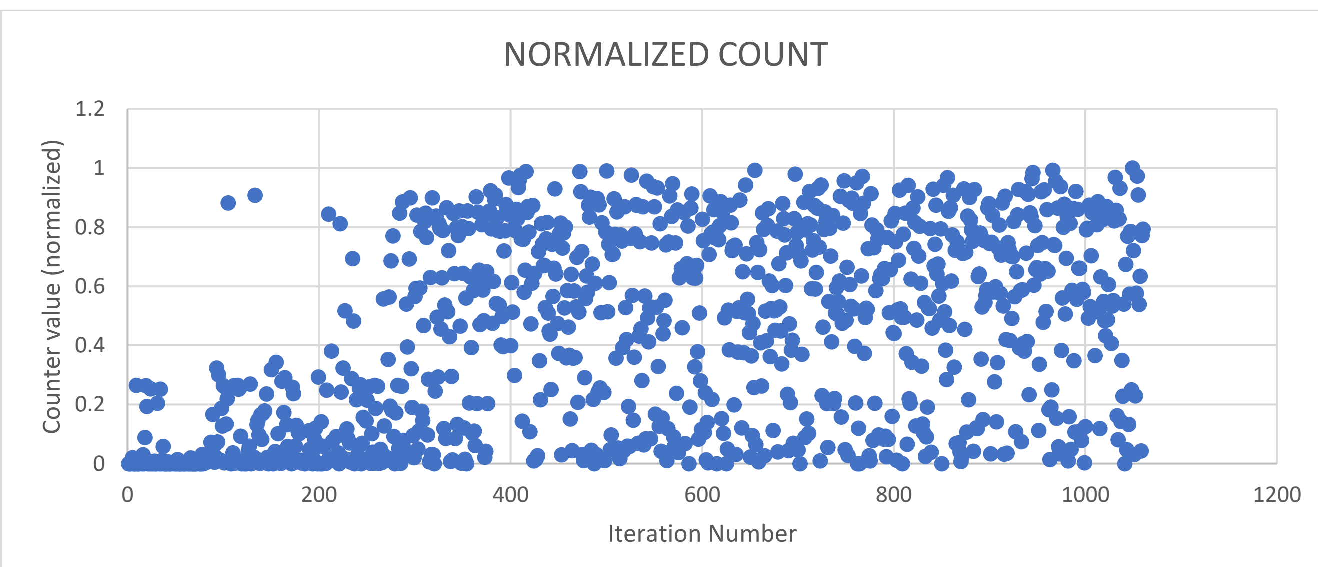

Figure 9: Scatterplot of outputs of Decision Tree Regressor with Retired Instructions, normalized.

Looking at figure 9, stability is reached after 280 samples (experiments) and continues there for the rest of the predetermined maximum number of experiments. Over time a higher density of points is observed near the max values around experiment 500.

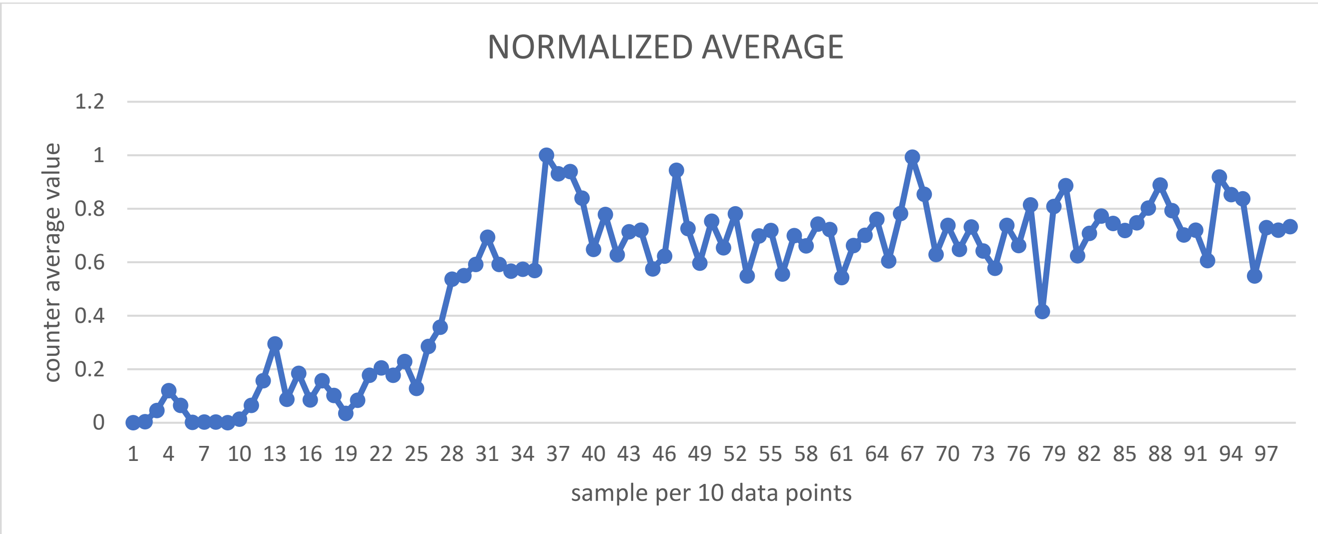

Figure 10: Outputs of Decision Tree Regressor with Retired Instructions, normalized, and averaged every 10 samples.

The average results in figure 10 show that the output value 'settles' around 0.75 units. This graph also shows very well how it raises from near zero values to the stability and reveals that while the algorithm found the stability value early, its certainty of that value being correct was not as high. This is why a slight increase in the average is seen soon after.

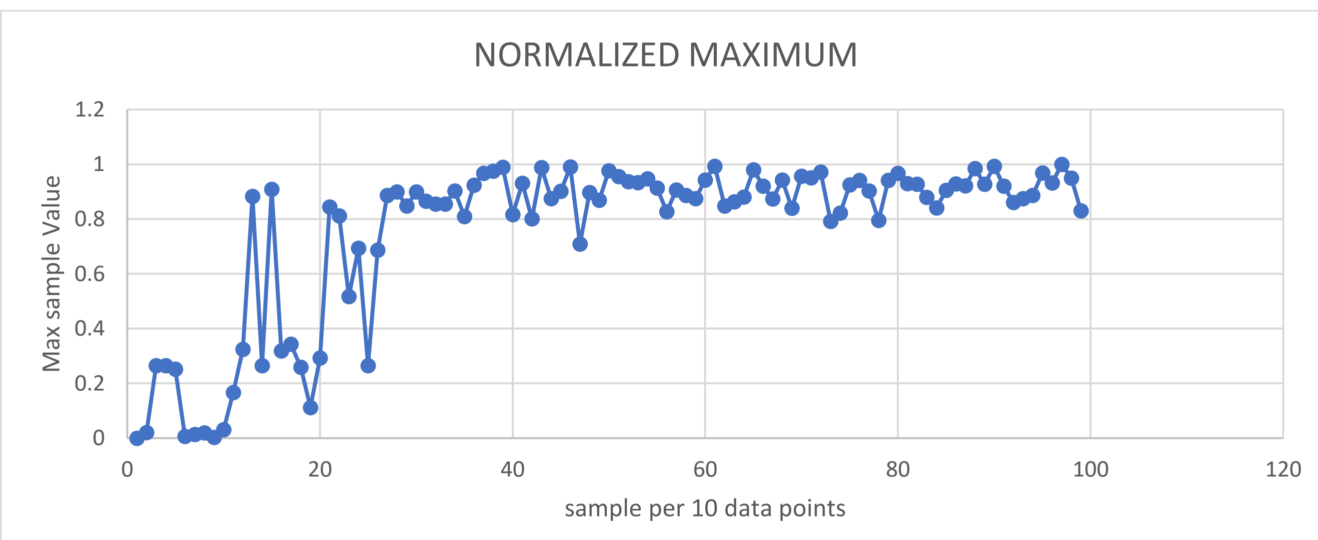

Figure 11: Outputs of Decision Tree Regressor with Retired Instructions, normalized, and showing the maximum of every 10 samples.

The curve shape shown by the maximum values of the graph shown in figure 11, clearly exposes a cleaner (with less variation) shape of how the algorithm slowly reaches the stable period around 0.95. There are two outlier points near the beginning (samples 105 and 133 from figure 9) probably caused by a streak of luck in the randomization, that also helped in giving the algorithm a hint of optimized parameters.

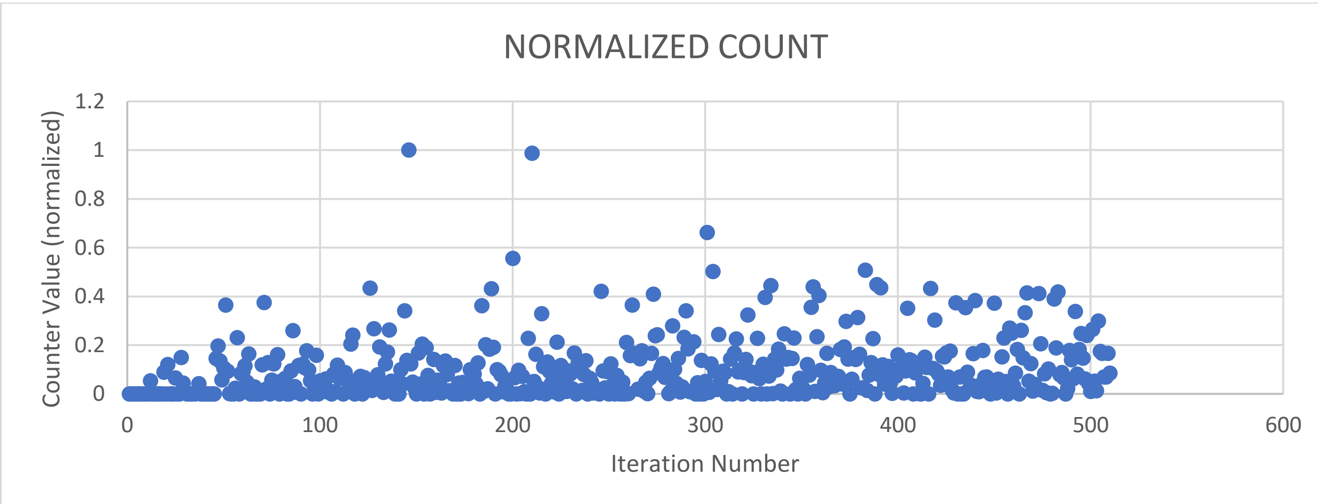

Figure 12: Scatterplot of outputs of Decision Tree Regressor with LLC victims, normalized.

In figure 12, a different counter shows that the algorithm struggles much more to reach any resemblance of stability at a higher range. However, it is still very clear that the algorithm did manage to achieve increasingly higher counter values but took over 300 iterations to do so. A certain convergence started to appear near iteration 460, but even so it did so with a disappointing low value.

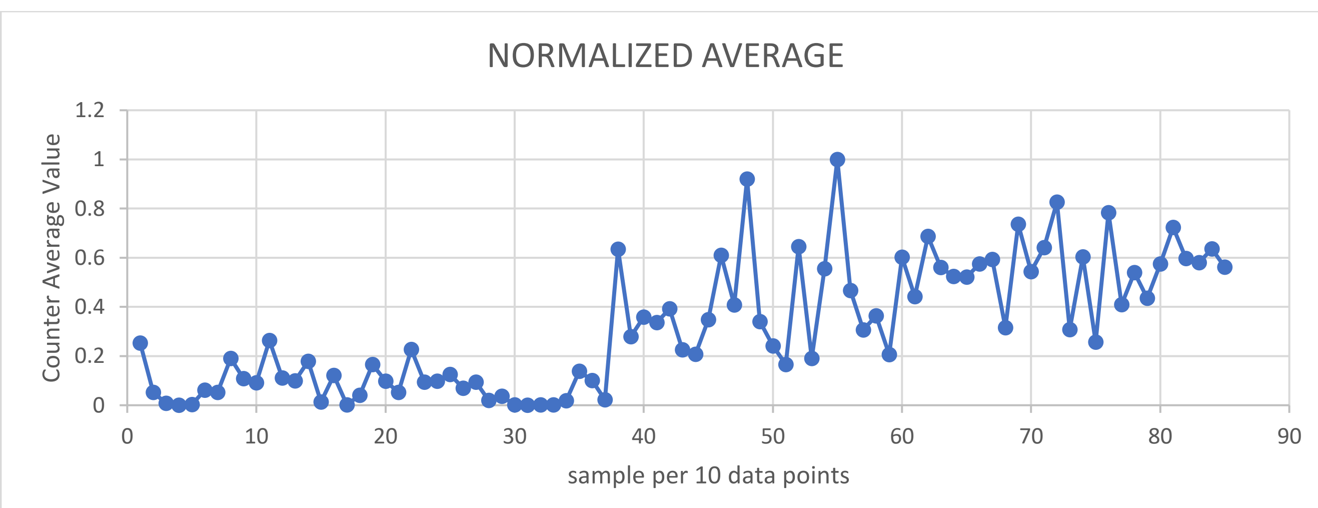

Figure 13: Outputs of Decision Tree Regressor with LLC victims, normalized, and averaged every 10 samples.

As shown in figure 13, there is a clear tendency in the average output to converge to a value of 0.6, but with much more oscillation in the average compared to the previous counter (fig 9 & 10).

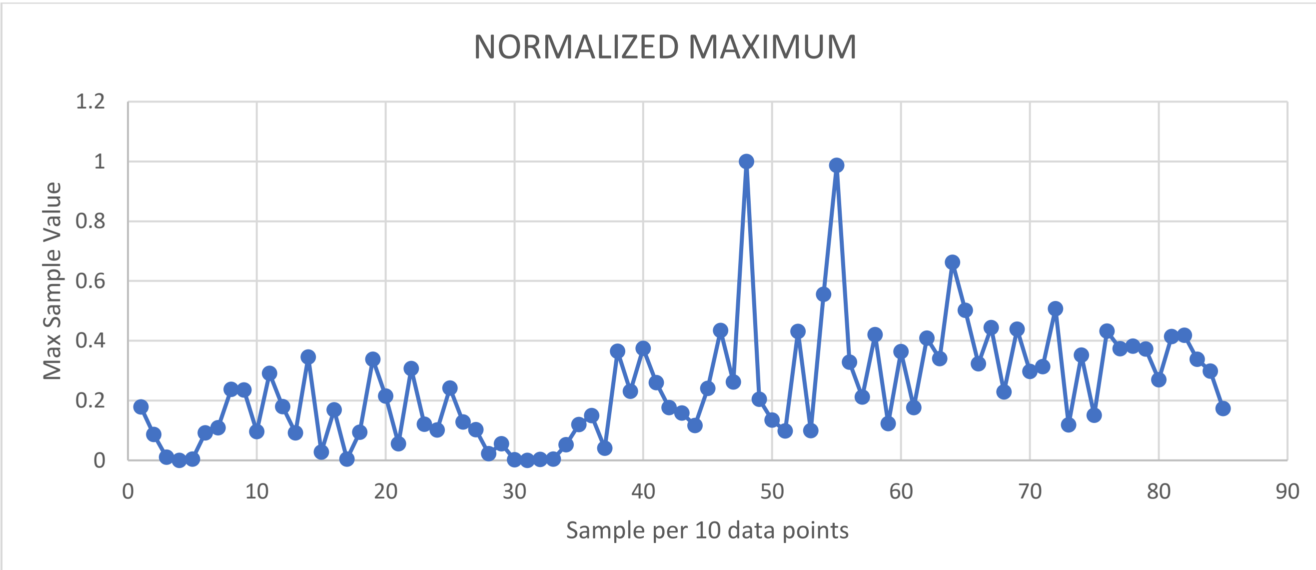

Figure 14: Outputs of Decision Tree Regressor with LLC victims, normalized, and showing the maximum of every 10 samples.

Figure 14 shows an almost identical behavior with slightly higher values, reflecting that for this case the overall model reflects very closely to the maximum values.

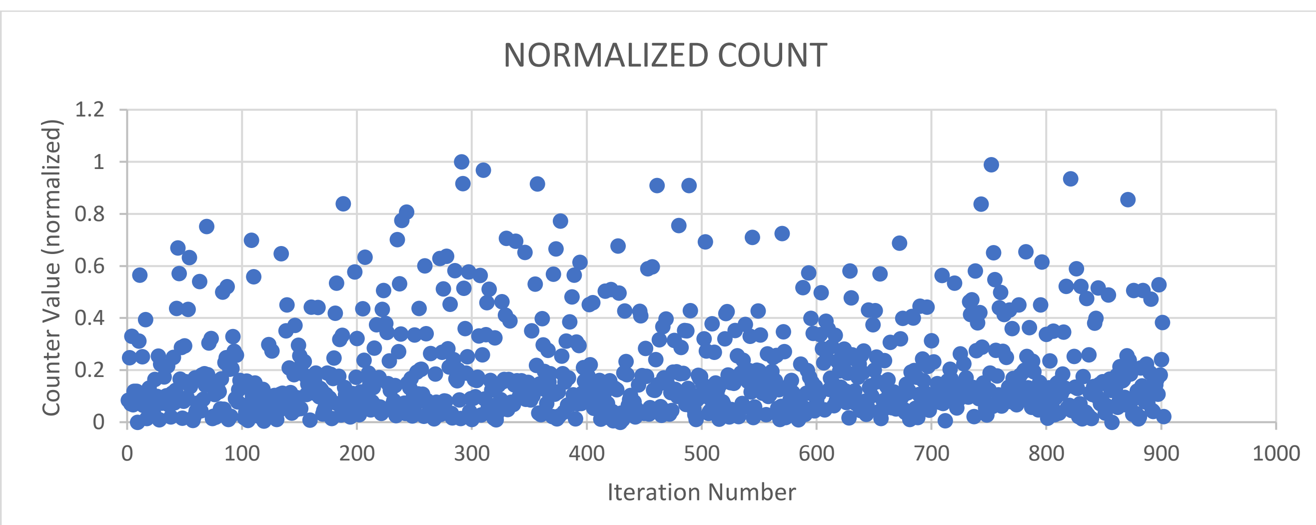

Figure 15: Scatterplot of outputs of Decision Tree Regressor with Internal PMON 1.

In figure 15 there is a clear tendency to a certain value, but the algorithm is struggling to achieve stability. There is a clear convergence between 0.6 and 0.8 but the algorithm gets a bit lost around iteration 450 and tries to converge to the same value again from there onwards.

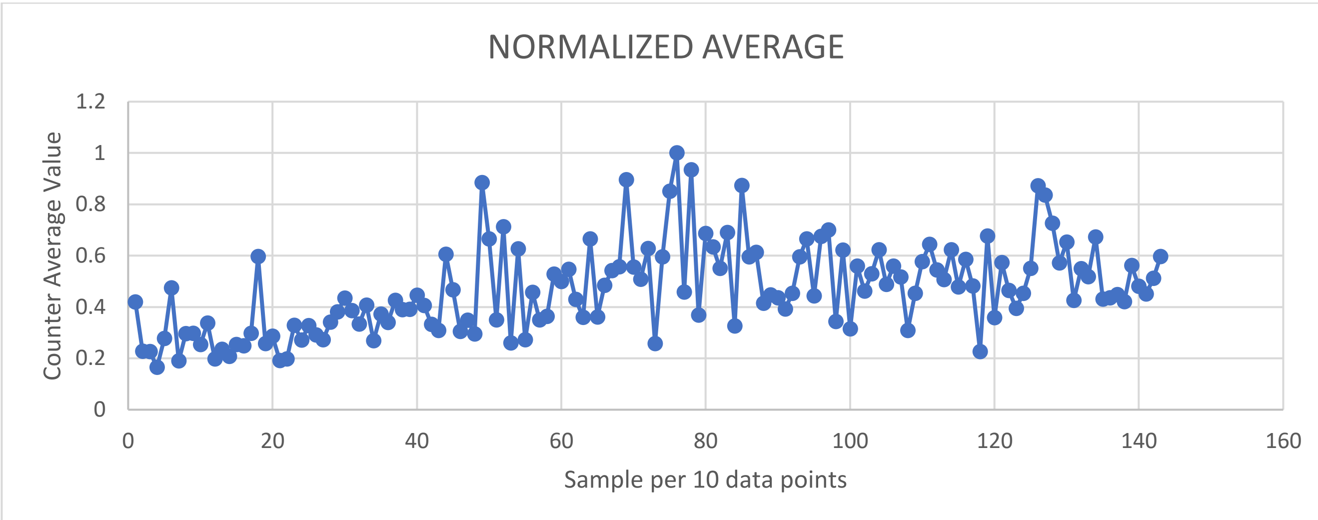

Figure 16: Outputs of Decision Tree Regressor with Internal PMON 1, normalized, and averaged every 10 samples.

Looking at the average now from figure 16 the tendency towards a value of 0.6 is clearly visible, but still with significant variation.

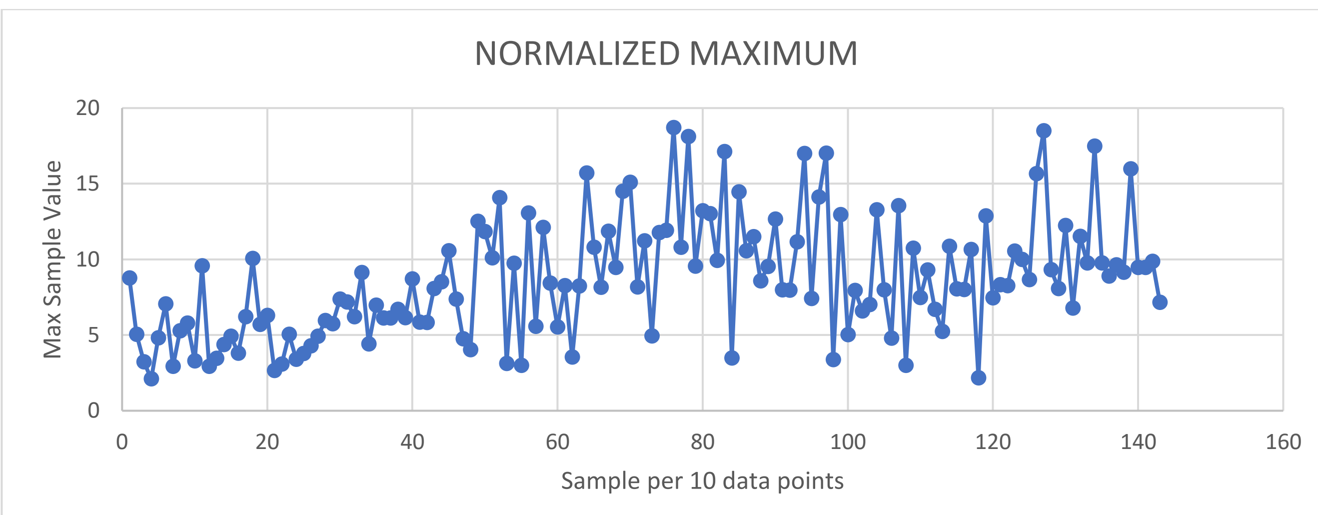

Figure 17: Outputs of Decision Tree Regressor with Internal PMON 1, normalized, and showing the maximum of every 10 samples.

Moving on to the maximums we can see basically the same behavior, with a clear tendency to a central value but with higher oscillations.

4.2 Random Forest Regressor

For the Random Forest algorithm, the preprocessor from the Decision Tree regressor was basically reused as-is, using the ML model from the sci-kit API from the ensemble code library, but this time with a much more complicated optimizer. Since the previous optimizer uses the data generated from each tree node to perform the analysis, that is not directly compatible with an ensemble of multiple Decision Trees in a Random Forest.

The optimizer was modified to take the values from each leaf in all trees and compare them between each other, and check for contradictions. In case a contradiction is found, it uses a voting arbitration system (of which if a draw is found, the decision is made at random). If a contradiction is not found, however, the values from the highest outcome leaf would be used.

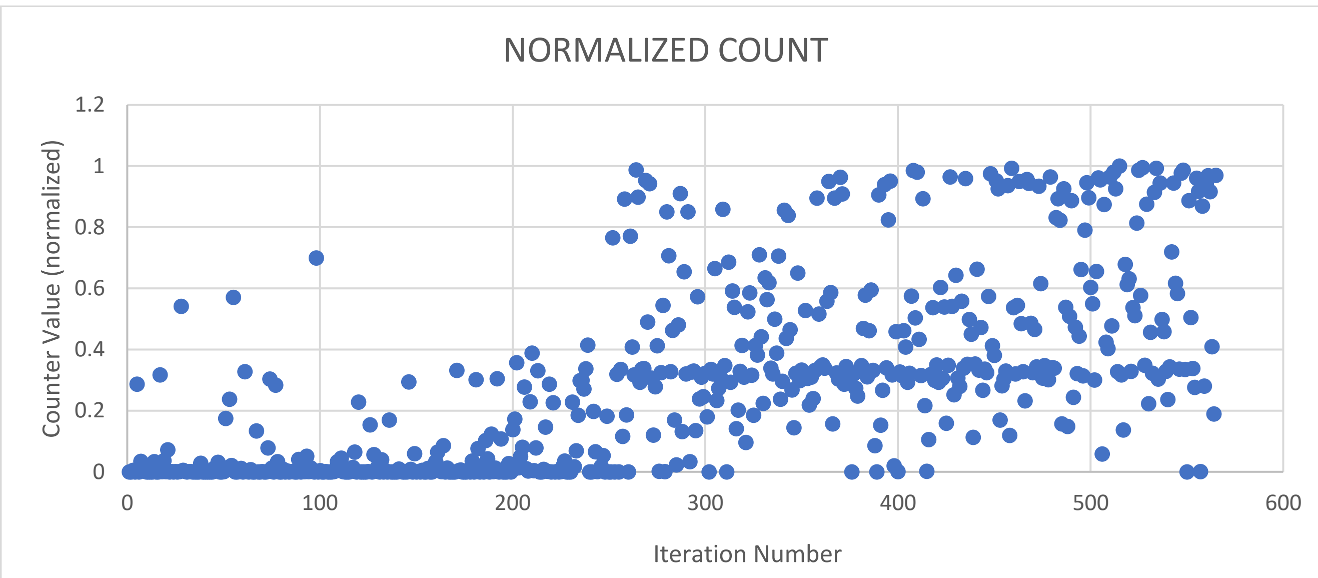

Figure 18: Scatterplot of outputs of Decision Tree Regressor with Retired Instructions, normalized.

In figure 18 the output has a significant rise around iteration 200 and stabilizes by iteration 270. A notable mention is that the lowest values (the bottom of the graph) is considerably higher than all other outputs until now by comparison.

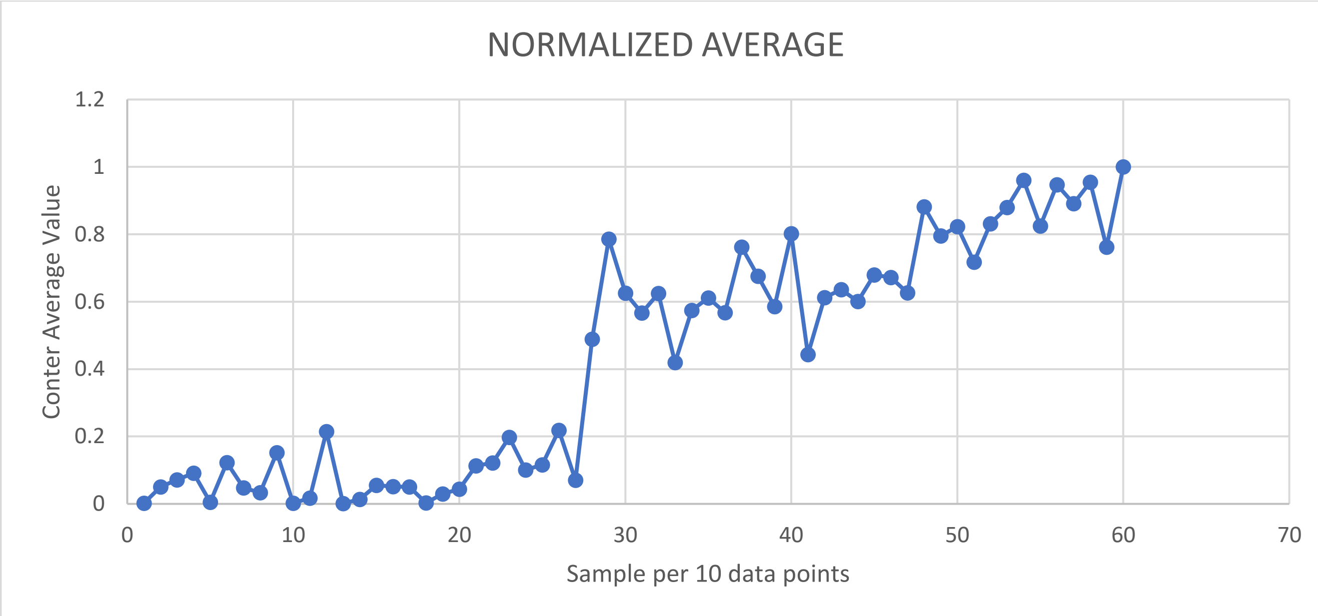

Figure 19: Outputs of Random Forest Regressor with Retired Instructions, normalized, and averaged every 10 samples.

Looking at figure 19, it is very clear that stability was reached around sample 30 but continued to climb a bit more until settling around a value of 0.8. It is also worth noting the relatively low variation perceived throughout the entire test.

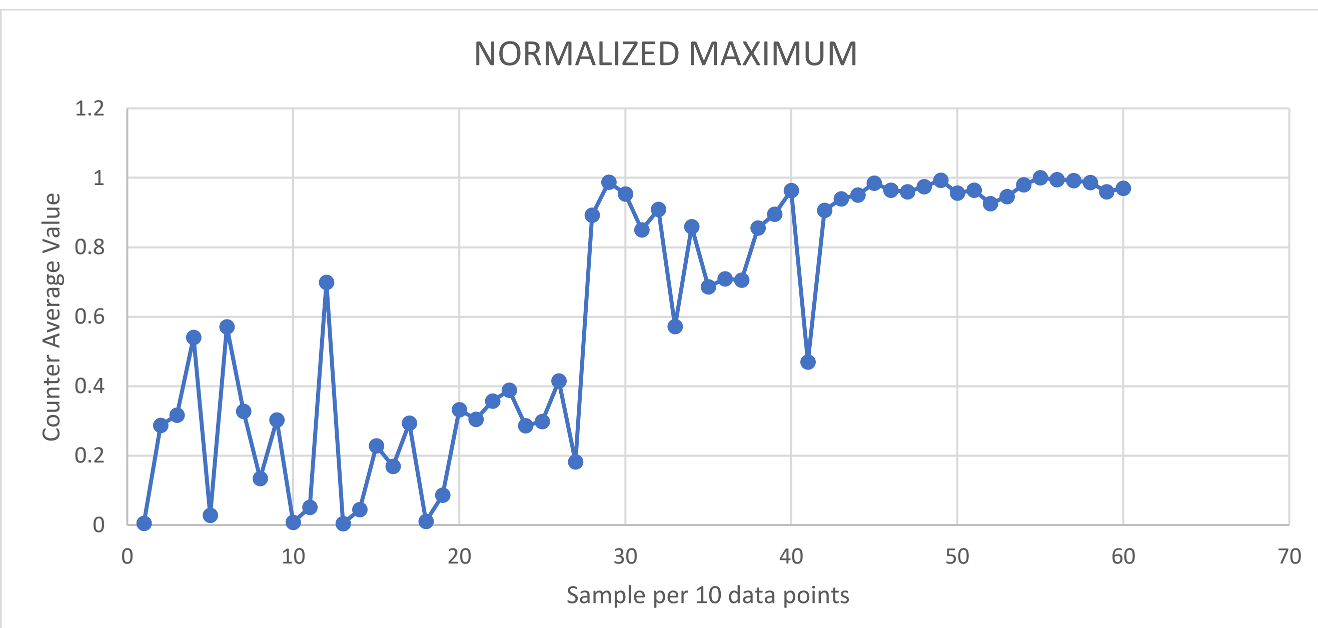

Figure 20: Outputs of Random Forest Regressor with Retired Instructions, normalized, and showing the maximum of every 20 samples.

Looking at figure 20, we can see the progression of the output value increase, keeping note of the difference in the scale of the graphs between figures 19 and 20.

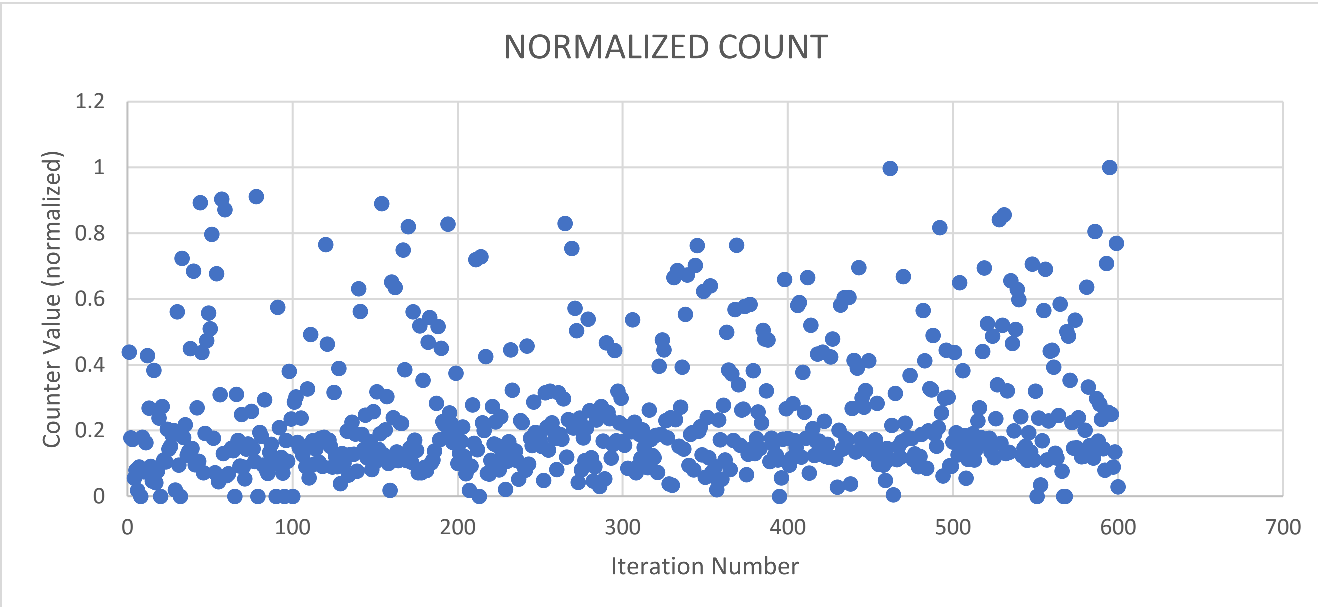

Figure 21: Scatterplot of outputs of Random Forest Regressor with LLC victims, normalized.

Changing to the next counter, figure 21 demonstrates that the algorithm is struggling more to achieve stability. A stable value is seen around 0.65 by iteration 162, but the algorithm still has low confidence of the parameters, however. By iteration 330 a significant increase in concentration of samples is seen with higher values, so this can be said to be the point where convergence is being reached.

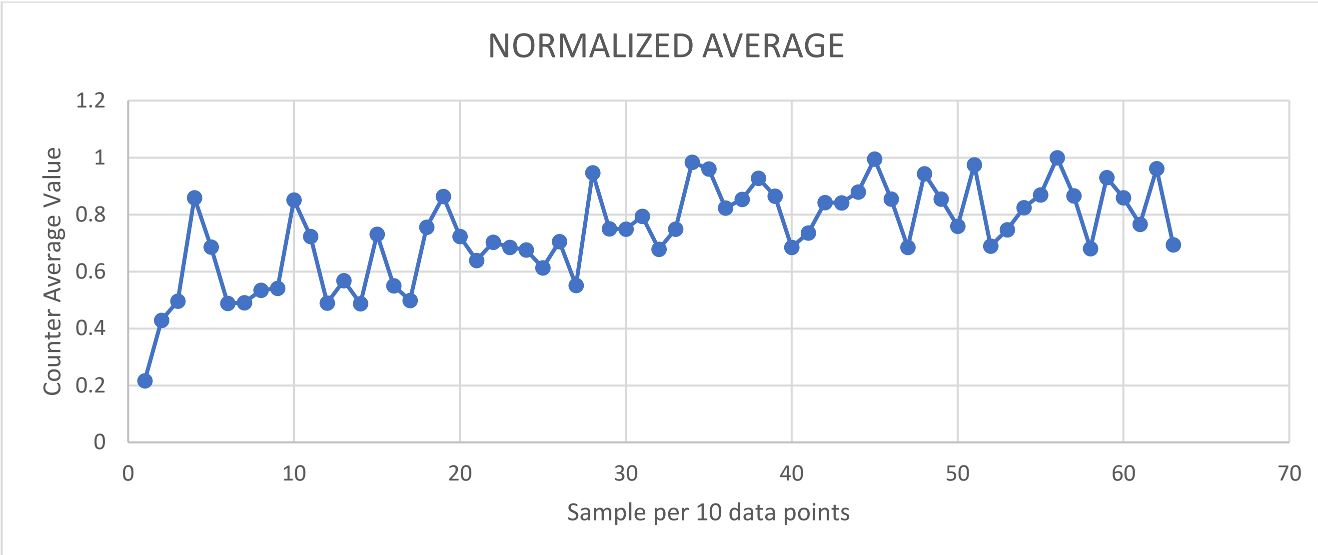

Figure 22: Outputs of Random Forest Regressor with LLC victims, normalized, and averaged every 10 samples.

Comparing figures 21 and 22, we can see that the convergence value is found really early, but the confidence of the algorithm is also extremely low, but increases over time, until settling around halfway through the test.

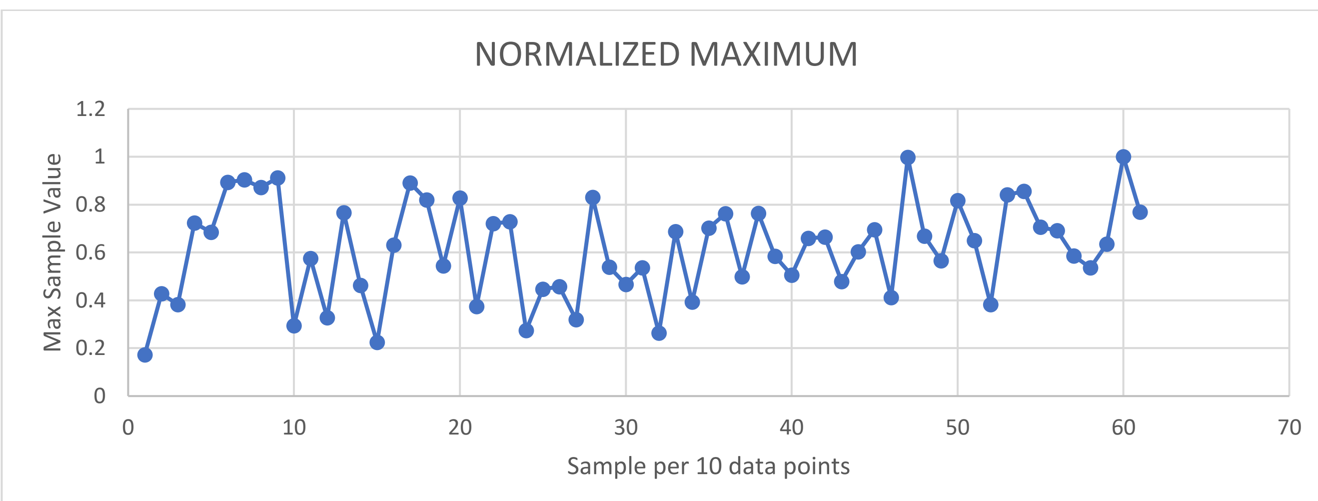

Figure 23: Outputs of Random Forest Regressor with LLC victims, normalized, and showing the maximum of every 10 samples.

Moving on to the max values, the top of the graph is reached very early on and is very often approached again, meaning that this test exemplifies a maximum-value priority case.

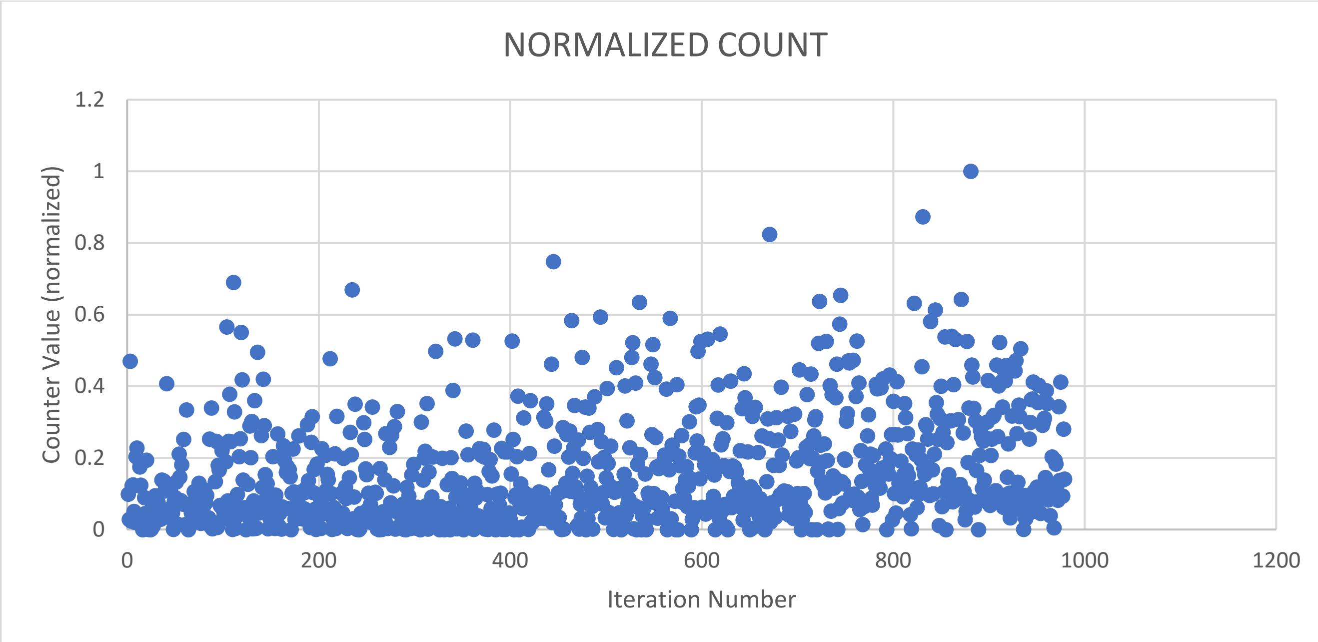

Figure 24: Scatterplot of outputs of Random Forest Regressor with Internal PMON 1, normalized.

Transitioning to the last counter, the algorithm managed to increase the output values almost immediately, reaching stability very early by the iteration 62 and remaining there for a while, but managing to find greater values by iteration 450, and converging to a new higher value.

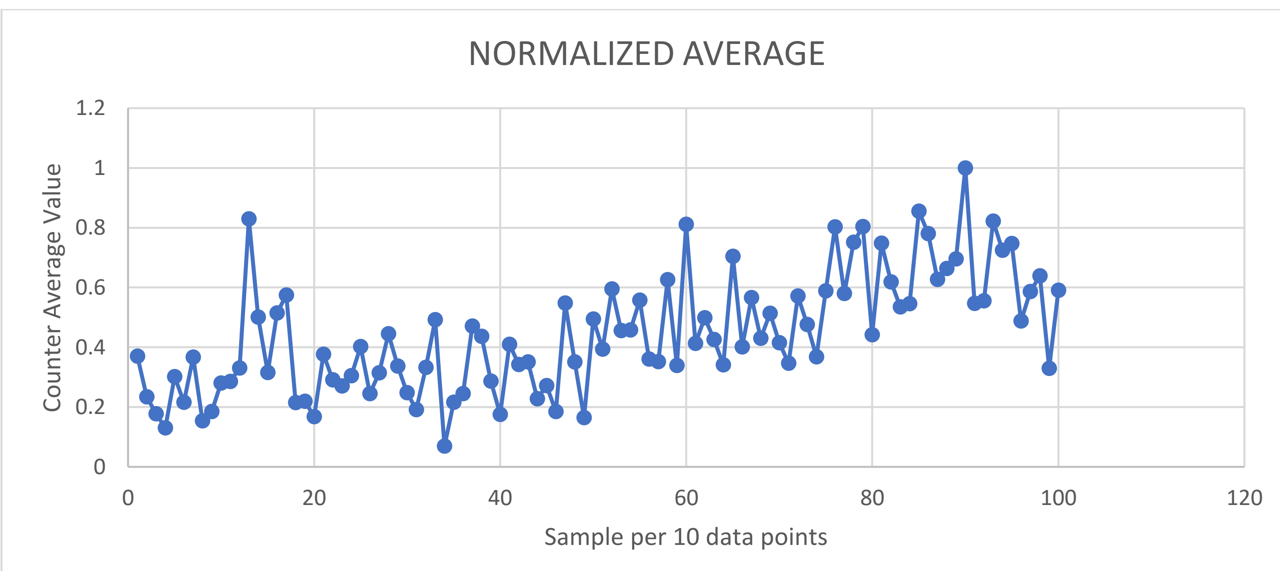

Figure 25: Outputs of Random Forest Regressor with Internal PMON 1, normalized, and averaged every 10 samples.

In figure 25 we see mostly the same, finding convergence very early in the test, but trend to higher values around halfway through and converging more to that value around iteration 75.

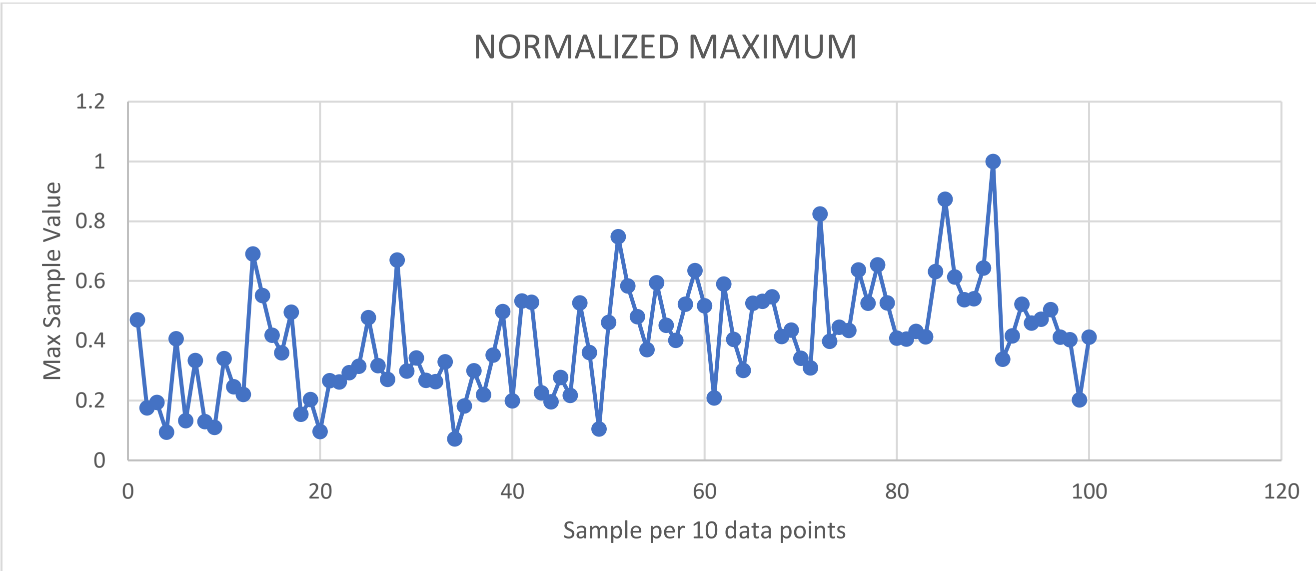

Figure 26: Outputs of Random Forest Regressor with Internal PMON 1, normalized, and showing the maximum of every 10 samples.

Finally, in figure 26, a similar case is seen, only this time the variation throughout is notably stronger. This also reflects that the algorithm is not fully converged, but still retains the upward tendency to a higher value.

It is worth mentioning that if this test were to be continued for longer, the upward tendency indicates that the possibility of reaching higher values and a higher convergence value is quite high.

4.3 Extra Trees

For the Extra Trees algorithm, the preprocessor from the Decision Tree regressor was basically reused as-is, using the ML model from the sci-kit API from the ensemble code library, and a slightly modified version of the Random Forest's optimizer.

The main difference between Random Forest and Extra Trees models is that Random Forest divides the total dataset in chunks to train each tree and creates an output taking the outputs of each tree into account, while the Extra Trees algorithm uses the entire dataset for all trees with slightly different parameters altered in a random matter. Taking this into consideration, the optimizer functions essentially in the same way, with modifications to be slightly more robust.

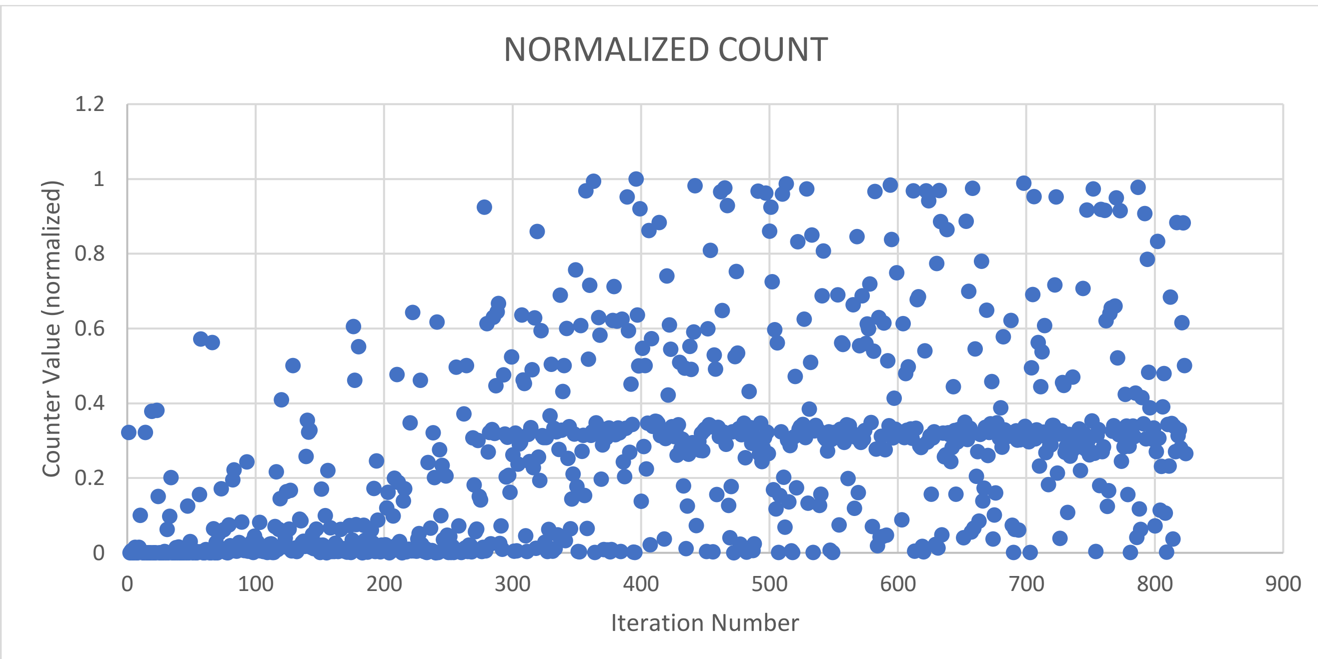

Figure 27: Scatterplot of outputs of Extra Trees Regressor with Retired Instructions, normalized.

Starting with figure 27, a very early rise in outputs is observed, but with very notable variation. Nevertheless, a continuous rise in the output is present from the very start until iteration 350, where stabilization is seen. There are also two parallel 'convergence lines' around values 0.33 and 0.98, that means some trees in the algorithm are trying to repeat experiments similar to those at counter value 0.33 and others are trying to repeat at counter value 0.98.

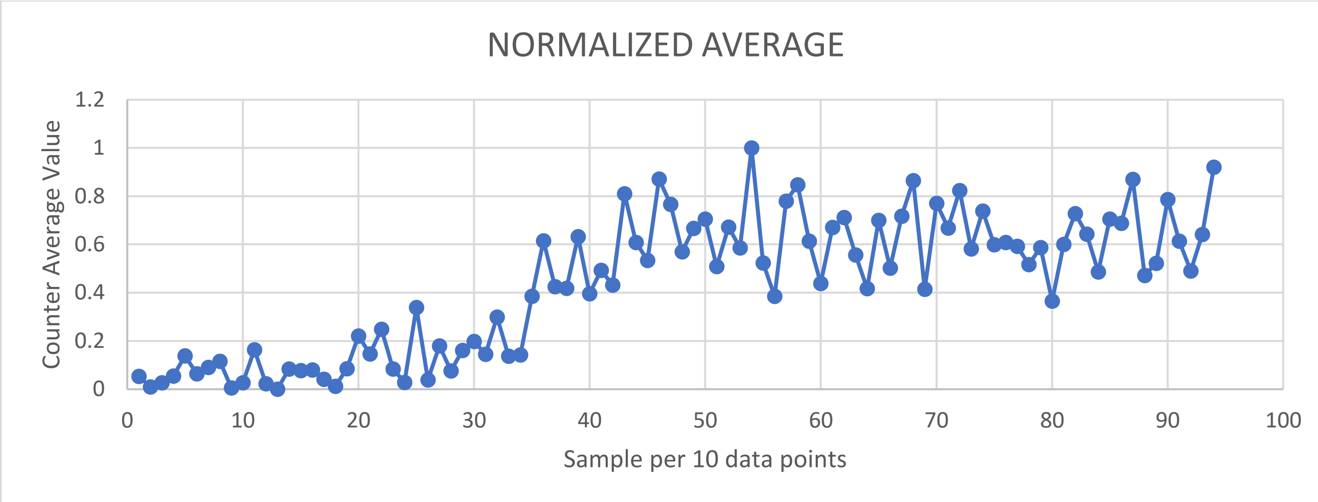

Figure 28: Outputs of Extra Trees Regressor with Retired Instructions, normalized, and averaged every 10 samples.

Continuing with figure 28, a clear tendency to stabilize around 0.6, with a notable deal of variation. The average does not seem to reflect much of the convergence towards the value of 0.98, meaning that there are still around half of values around 0.3 (average of 0.3 and 0.9).

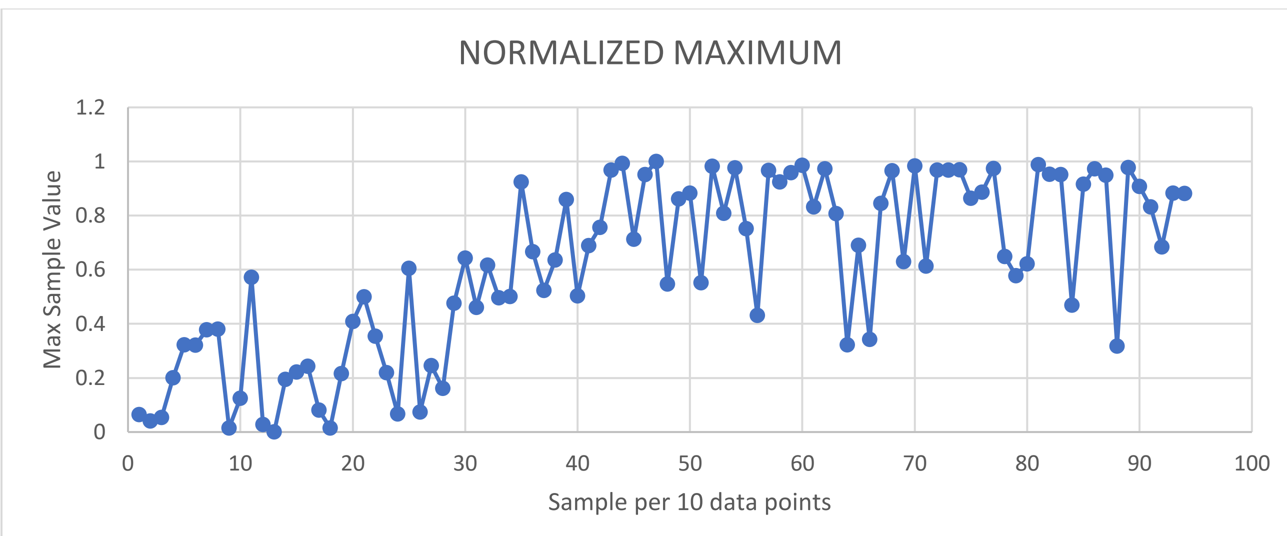

Figure 29: Outputs of Extra Trees Regressor with Retired Instructions, normalized, and showing the maximum of every 10 samples.

Moving to figure 29, shows a very consistent value hovering around 0.98, meaning the algorithm is very good a replicating high output values. The samples under 0.4 after iteration 60, show the tendency to repeat experiments in the 0.33 convergence band as well.

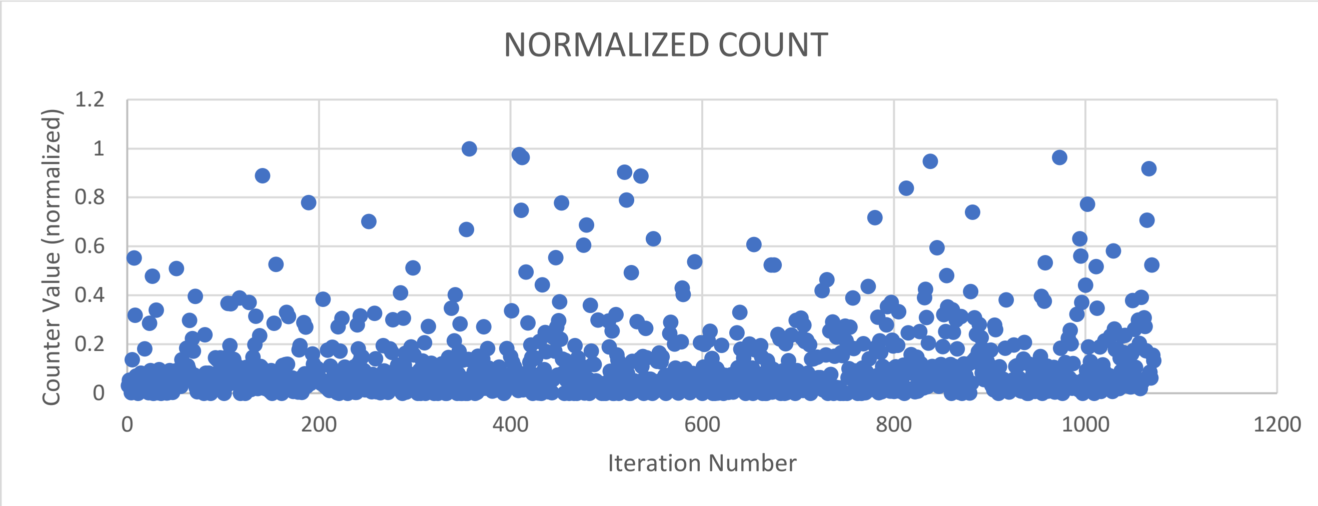

Figure 30: Scatterplot of outputs of Extra Trees Regressor with LLC victims, normalized.

Changing the counter to LLC victims shows remarkably increased variations, and the lack of convergence for this test. That is not to say that a higher value was never reached, but the algorithm failed to latch on to the correct parameters, thus far. It is highly likely that if left for more iterations a convergence point may have been reached as higher values will be repeating consistently.

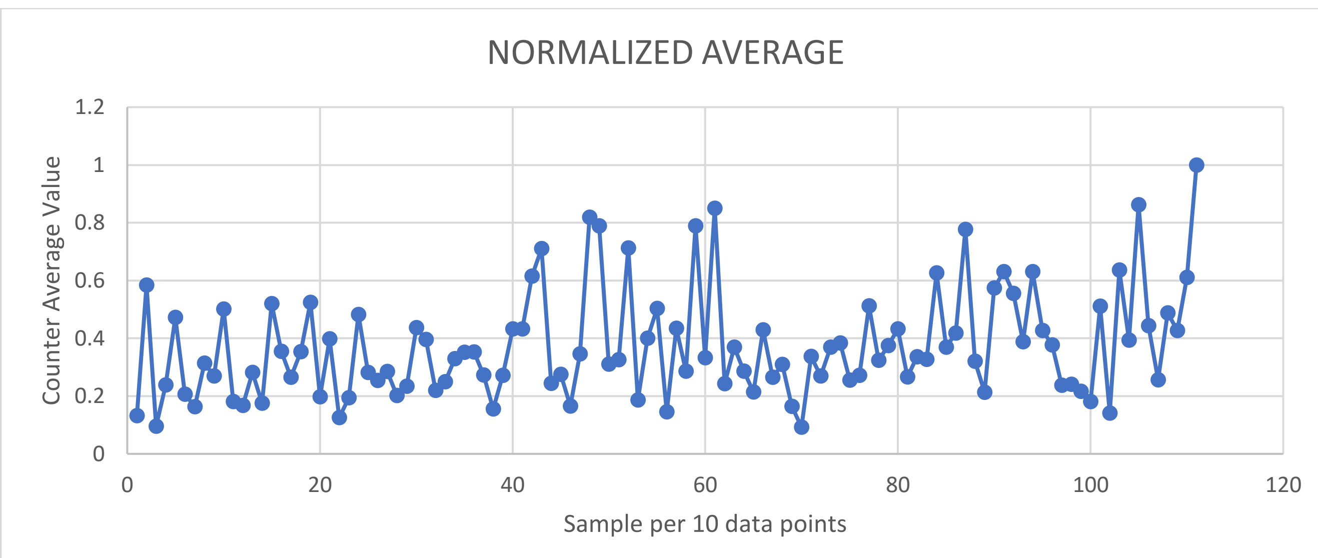

Figure 31: Outputs of Extra Trees Regressor with LLC victims, normalized, and averaged every 10 samples.

Figure 31 shows the same erratic behavior, reflecting the lack of convergence once more. There is, however, a resemblance of a tendency in the last samples, reinforcing the theory of possible convergence if left to run for more iterations.

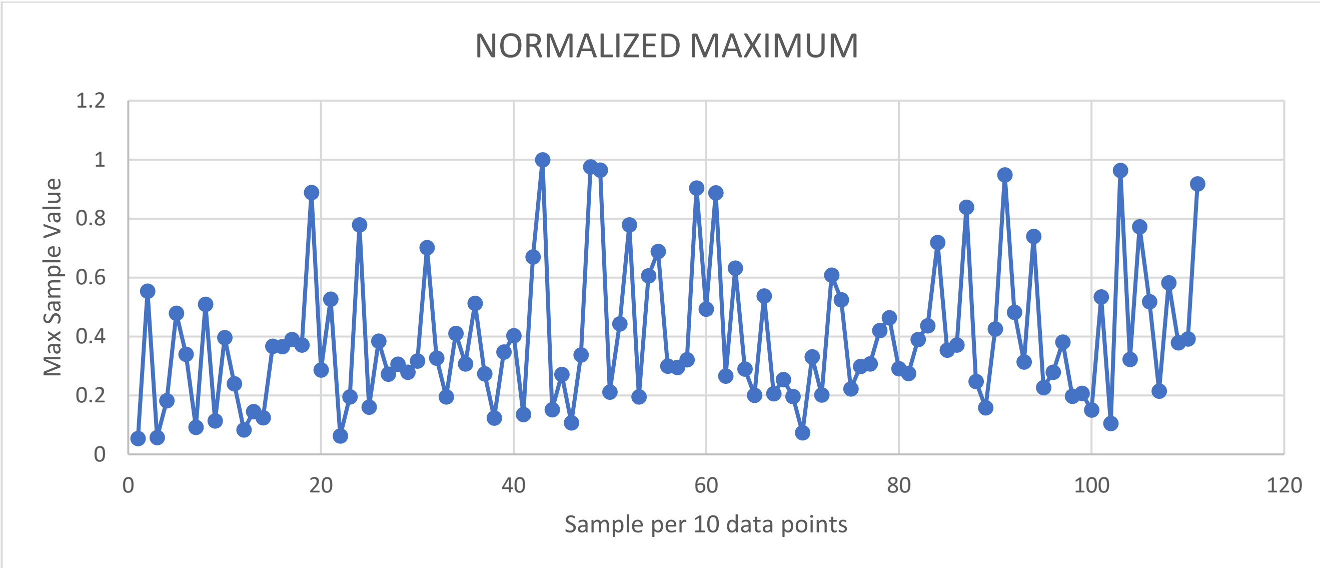

Figure 32: Outputs of Extra Trees Regressor with LLC victims, normalized, and showing the maximum of every 10 samples.

Figure 32 is seemingly more chaotic, but it is in reality only the result of high values caused by the randomization and the efforts of some leaf values that did not get many repetitions.

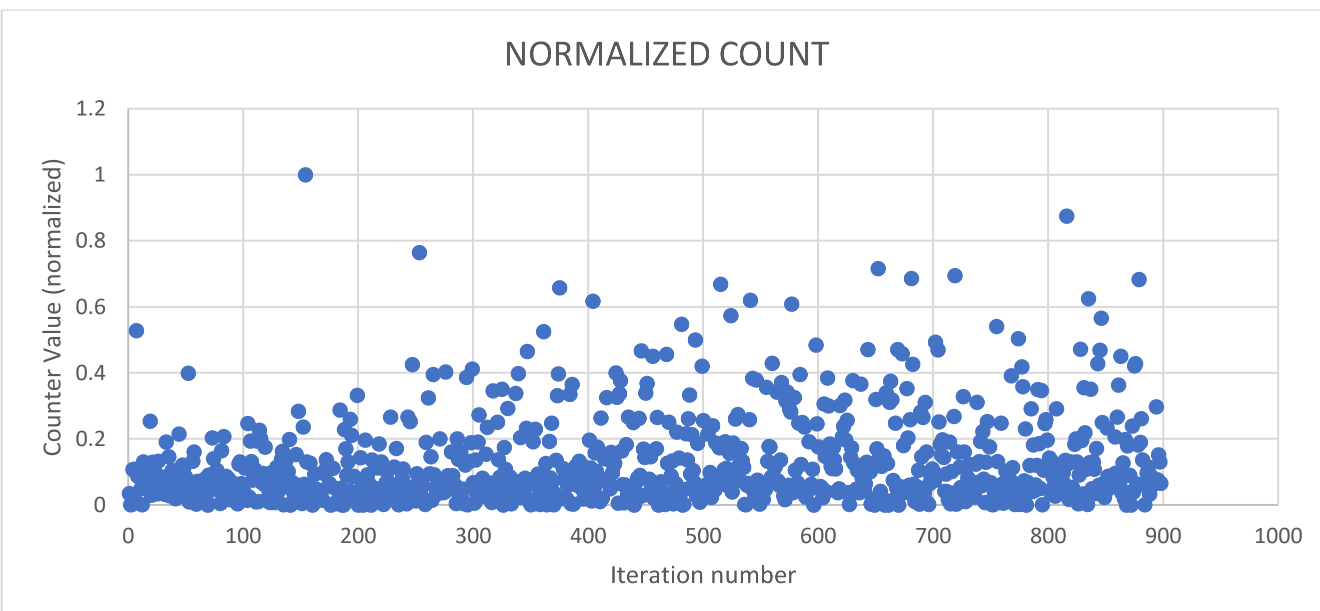

Figure 33: Scatterplot of outputs of Extra Trees Regressor with Internal PMON 1.

Last but not least, the Internal PMON 1 counter shows a much more stable behavior during the test, allowing the algorithm to slowly manage increasingly outcomes and showing tendency to convergence by iteration 600, around the 0.4 counter value.

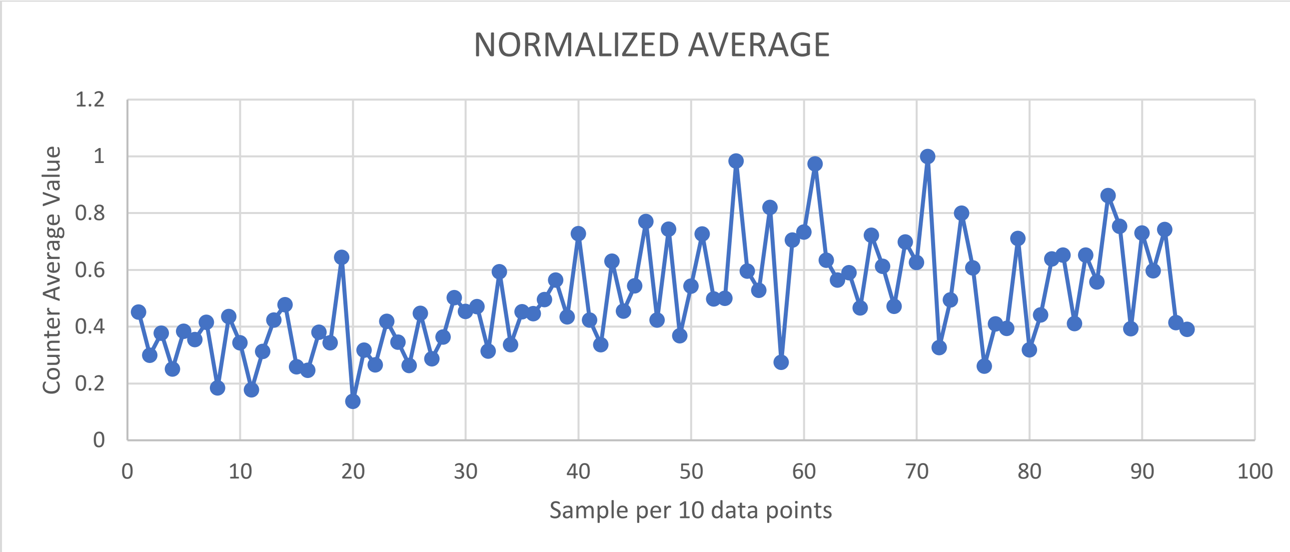

Figure 34: Outputs of Extra Trees Regressor with Internal PMON 1, normalized, and averaged every 10 samples.

The average in figure 34 shows a clear tendency to the increased values and settling around 0.65, but also reveals more variance than originally thought, decreasing slightly by the end.

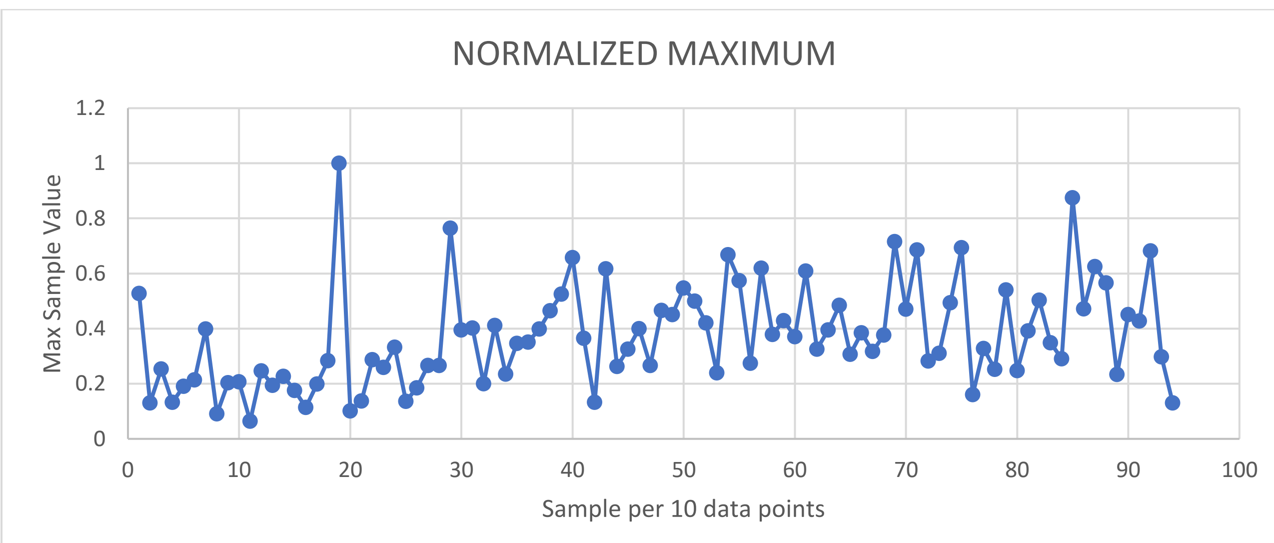

Figure 35: Outputs of Extra Trees Regressor with Internal PMON 1, normalized, and showing the maximum of every 10 samples.

Figure 35 shows notorious variation that slowly decreases but remains quite big all the way to the end. This is a sign that the algorithm is still trying to stabilize and will get better if left to run for more iterations.

4.4 KNN

The concept of this algorithm is to greatly improve the pre-processor stage of the algorithm by the use of clustering. Instead of passing the entire dataset to the ML model to process, an implementation of KNN was made to group the nearest data points to the maximum value found at any given moment and use that slightly reduced dataset (now generated by the output from KNN Scikit model) to train the Decision Tree Regressor (DTR) ML model.

The Decision Tree model and optimizer used for this were exact copies from the first algorithm in this chapter, just modified so that it now works with the output of the KNN pre-processor.

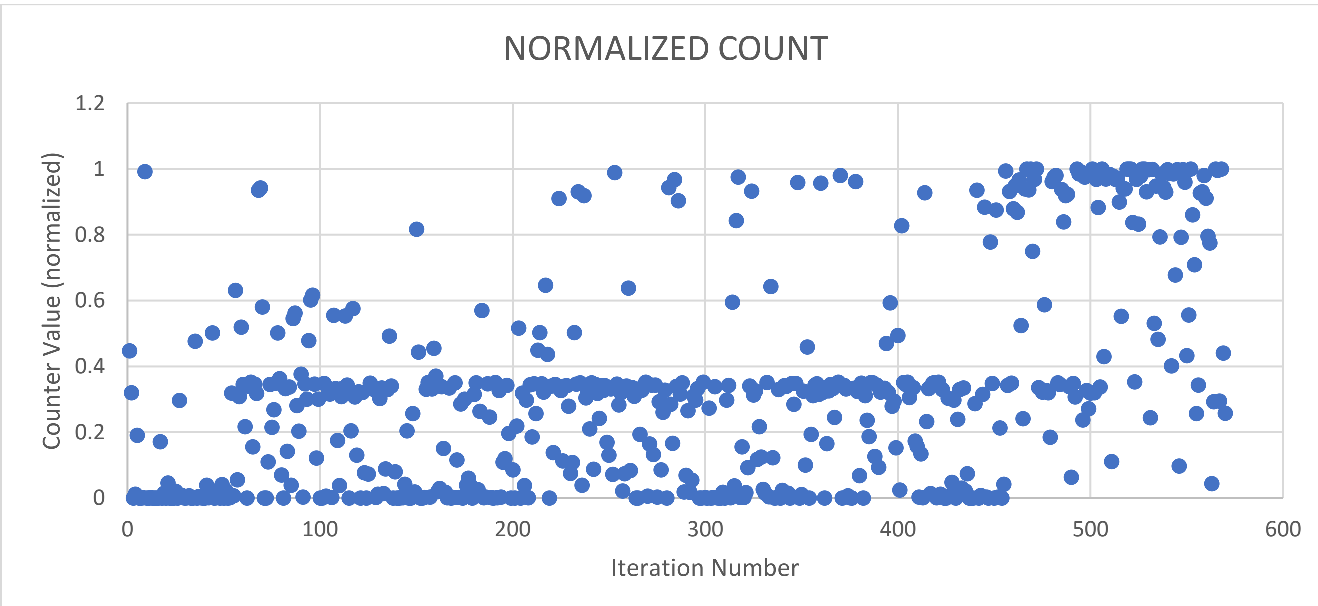

Figure 36: Scatterplot of outputs of KNN + DTR with Retired Instructions, normalized.

With the KNN algorithm overall we see a big tendency to 'snap' or 'jump' between different lines of convergence, instead of gradually transitioning from one to the other. Figure 36 is a great example of this, as can be observed by the jumps on iterations 60 and 450, going from almost 0, to 0.35 to 1.

The KNN clustering is the most likely suspect of this behavior, since now the Decision Tree model is able to see a subset of the highest values at any given time, so once enough 'higher-than-the-current-point' accumulate, the algorithm tends to 'jump' to said value instead.

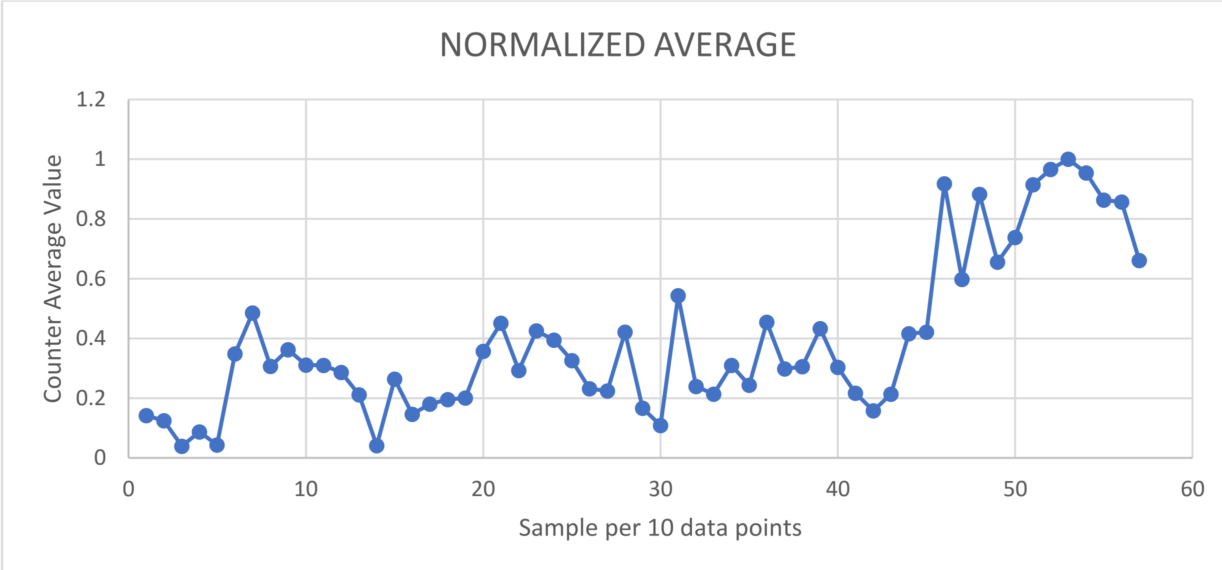

Figure 37: Outputs of KNN + DTR with Retired Instructions, normalized, and averaged every 10 samples.

Similar behavior is observed in the average, with two separate lines of convergence, at 0.35 and 0.9. Variation is still present though, product of randomization that is still mixed in.

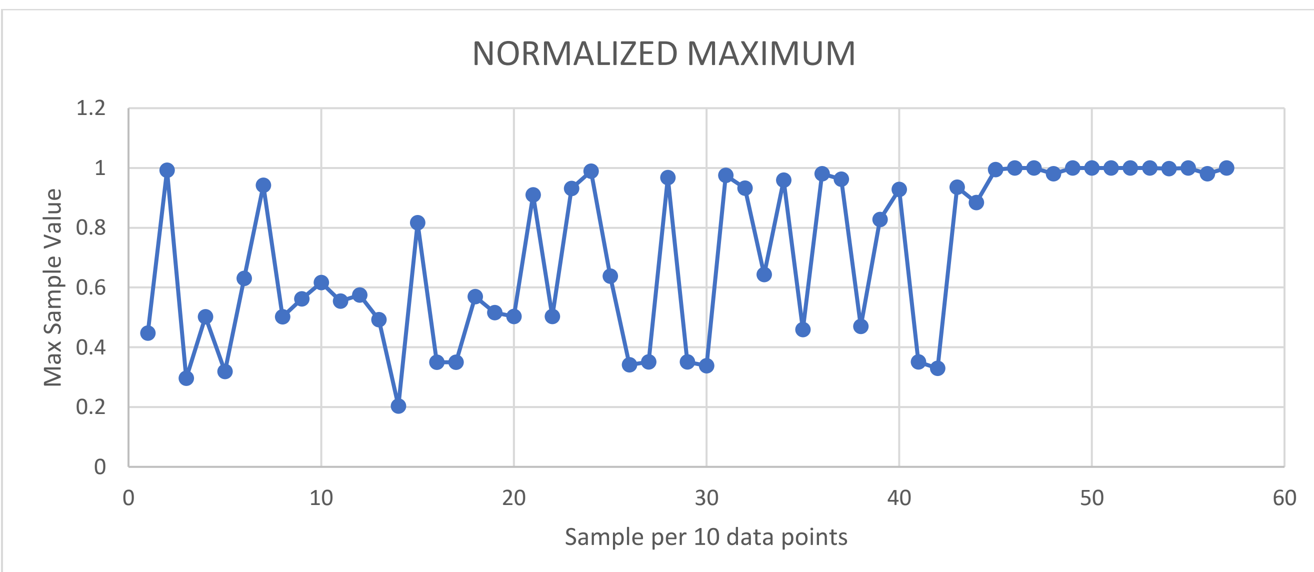

Figure 38: Outputs of KNN + DTR with Retired Instructions, normalized, and showing the maximum of every 10 samples.

Maximum values in figure 38 does how something interesting though, toward the end of the test, after the second 'jump' the maximum values are near perfect flat line, meaning that once the absolute maximum value is reached, the algorithm is really good at maintaining it.

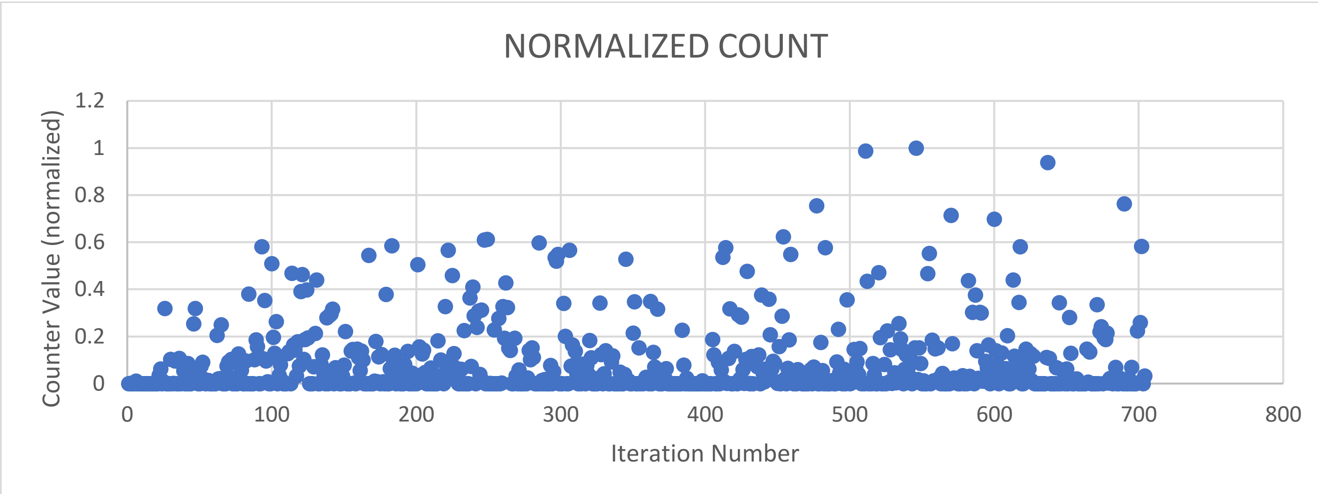

Figure 39: Scatterplot of outputs of KNN + DTR with LLC victims, normalized.

Moving to the next counter, we can see a similar behavior as before, just notably less obvious. The algorithm tries to first converge to the value of 0.6, and then finds some new higher values around 0.9, but fails to find enough to make the transition, stagnating for longer.

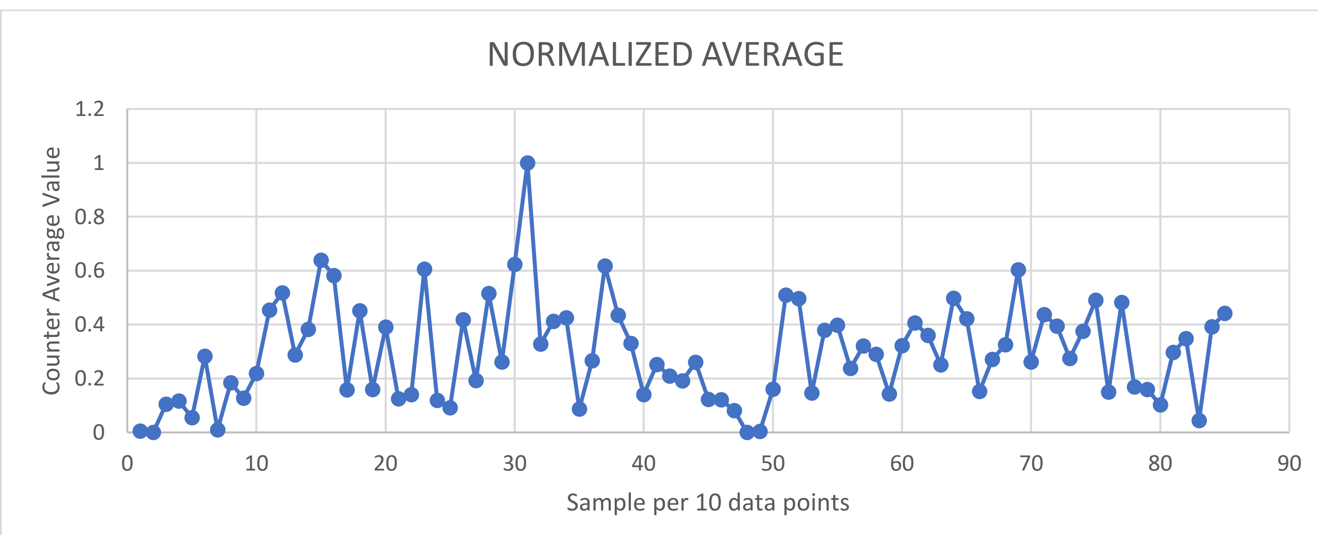

Figure 40: Outputs of KNN + DTR with LLC victims, normalized, and averaged every 10 samples.

Since the algorithm could not find more of the higher values, most of the samples remained in the lower values, reflected on figure 40. A lot of variation is still present, which can be seen in figure 41, indicator that the algorithm is still exploring within the observed values.

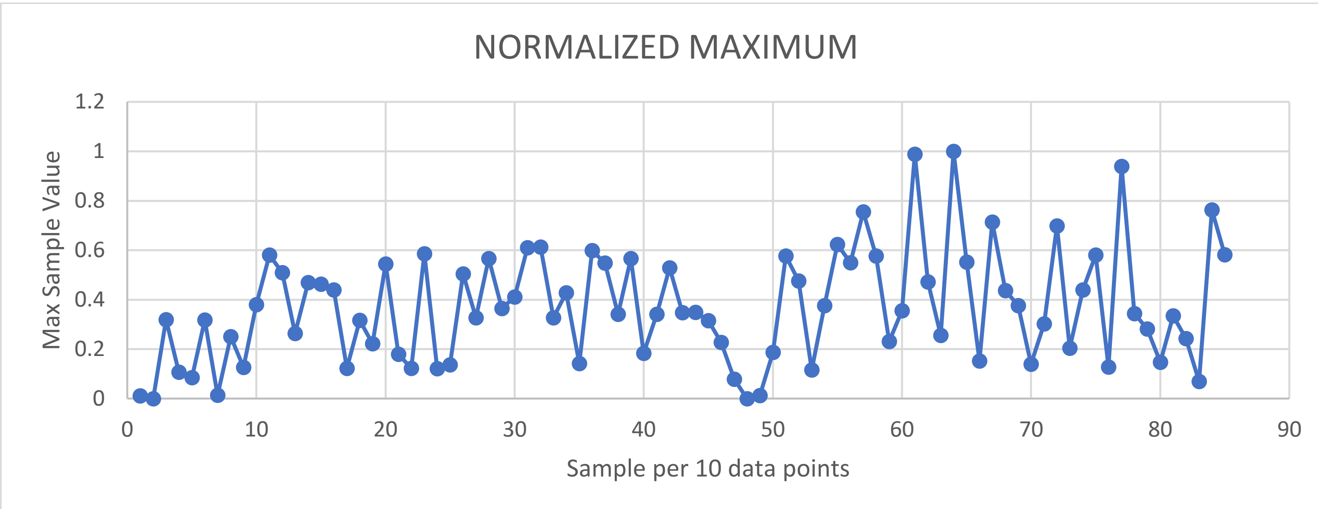

Figure 41: Outputs of KNN + DTR with LLC victims, normalized, and showing the maximum of every 10 samples.

Looking at the maximum values (figure 41) we can see an increase in variation, as well as the values of the samples in the second half of the test, meaning that the algorithm managed to find higher maximum values and was trying to replicate those, but failed to converge within the time of the test.

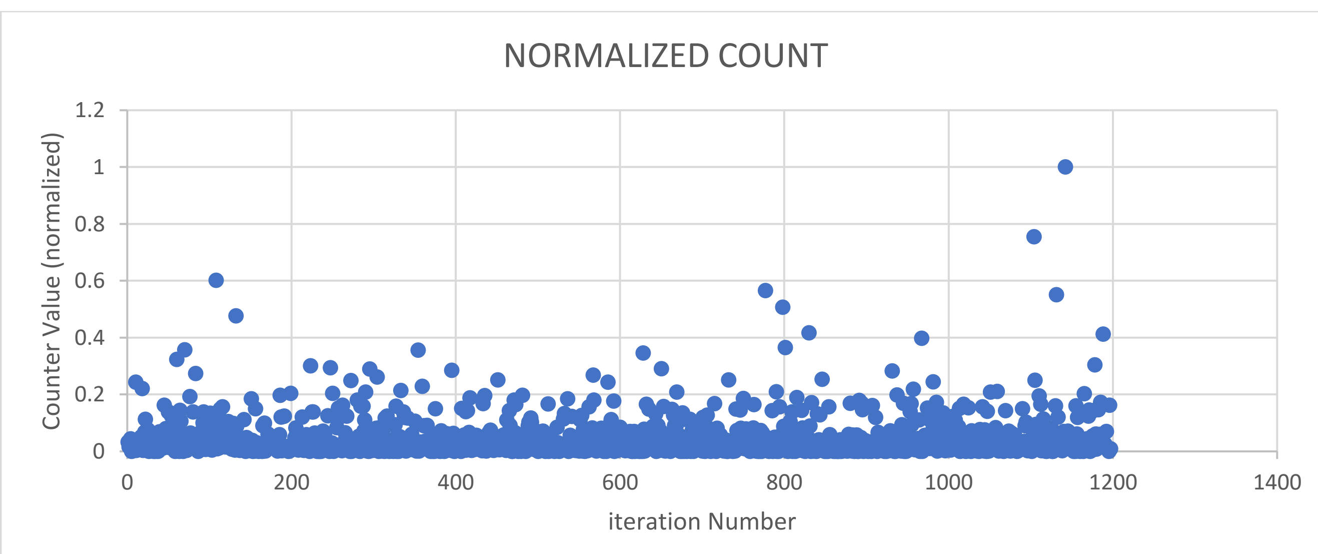

Figure 42: Scatterplot of outputs of KNN + DTR with Internal PMON 1, normalized.

Moving on to the last counter, it is clear that the algorithm struggled with this PMON the most, since no clear convergence line was found. There is some tendency to a higher value towards the end of the test, around iteration 1100, but this was already taking longer time than expected by the predefined set of tests to be run.

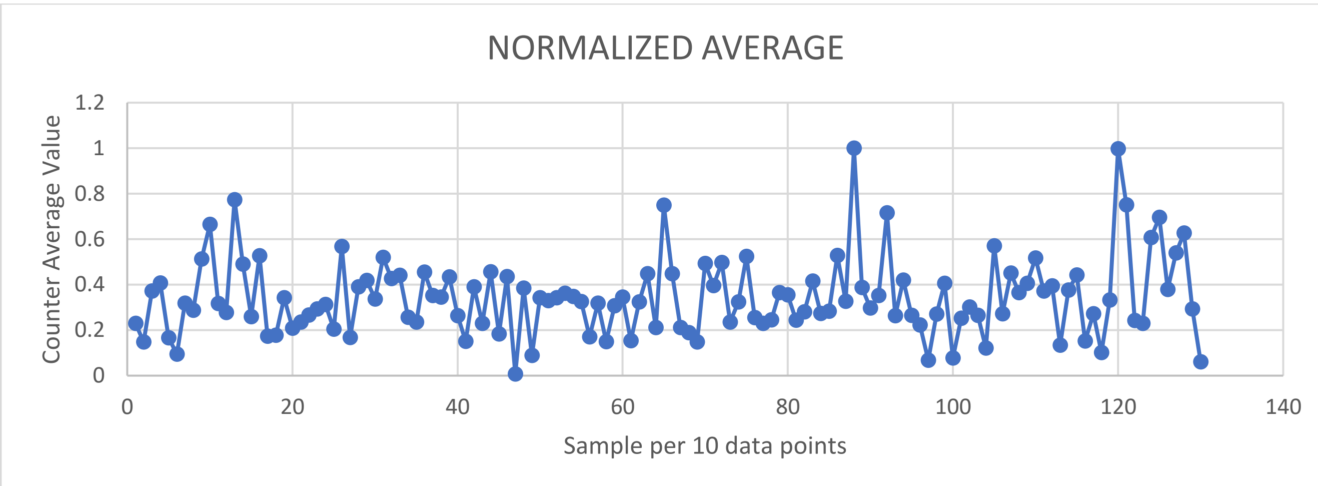

Figure 43: Outputs of KNN + DTR with Internal PMON 1, normalized, and averaged every 10 samples.

Average values in figure 43 reflect big amounts of variation, indicating again that the algorithm is still trying to find the correct parameters.

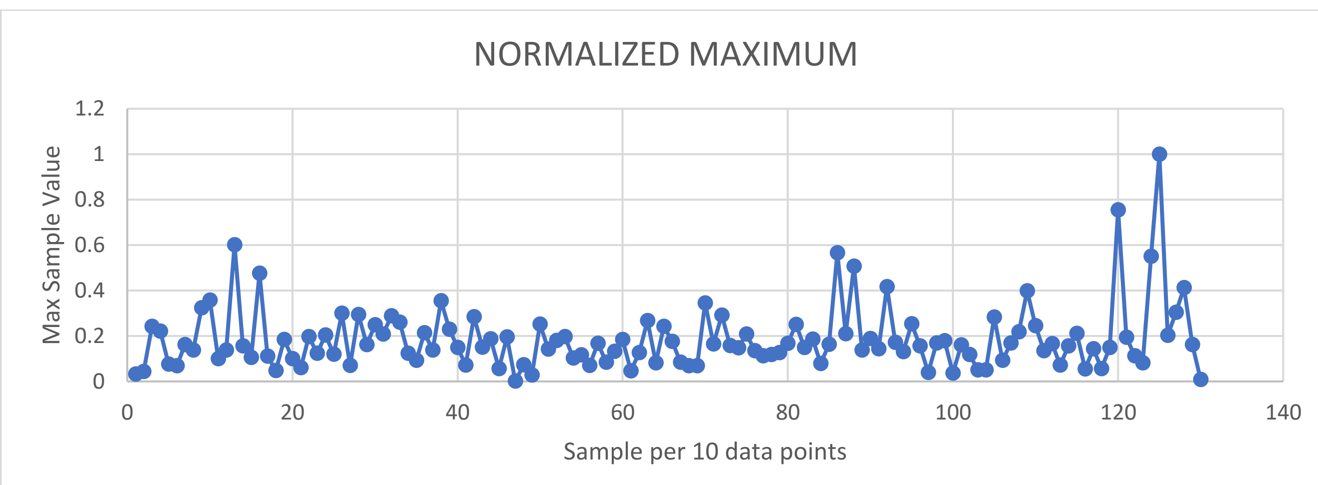

Figure 44: Outputs of KNN + DTR with Internal PMON 1, normalized, and showing the maximum of every 10 samples.

Figure 44 shows one more thing in addition to the average from figure 43: the big spikes toward the end, further reinforcing the theory of a possible convergence if left to run for more iterations.

4.5 Neural Network

A simple Neural Network implementation was already developed as a proof of concept for Intel's purposes before the start of this project and was adapted to fully work with the tests in the benchmarking, with some caveats.

While the software is capable of training, it is only capable of following a PMON for multiple cores (for all others the system is intended to focus on only one in order to isolate variables) and has a fixed period of 'retraining' of 30 iterations.

This means that while the results cannot be compared directly in some ways to the other algorithms (such as the maximum values), general behavior (such as convergence speed) is still directly comparable.

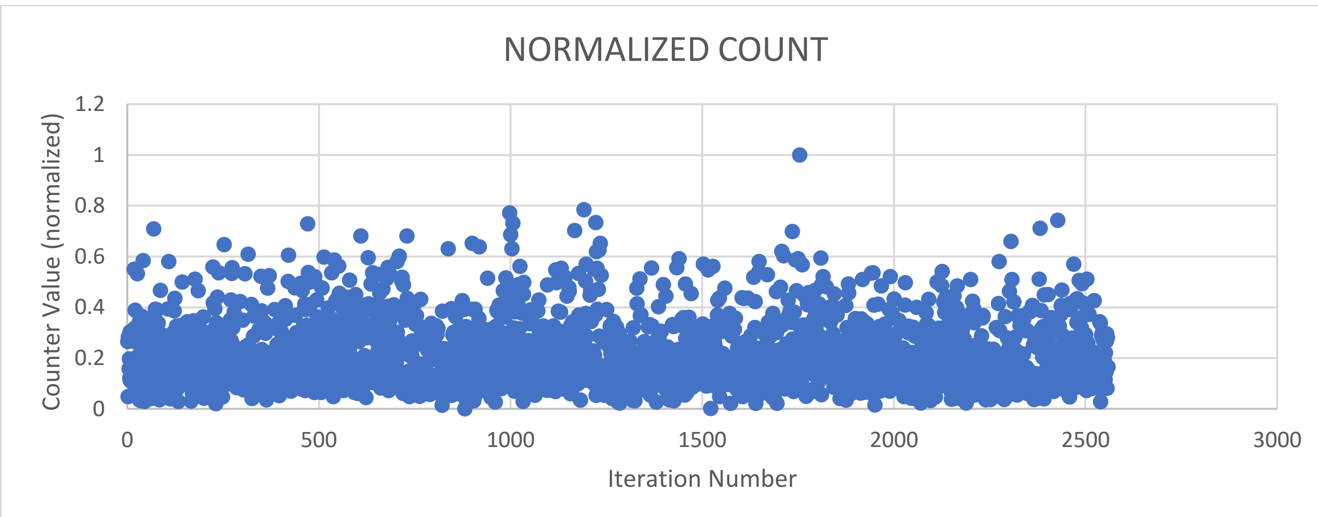

Figure 45: Scatterplot of outputs of a Neural Network with Retired Instructions, normalized.

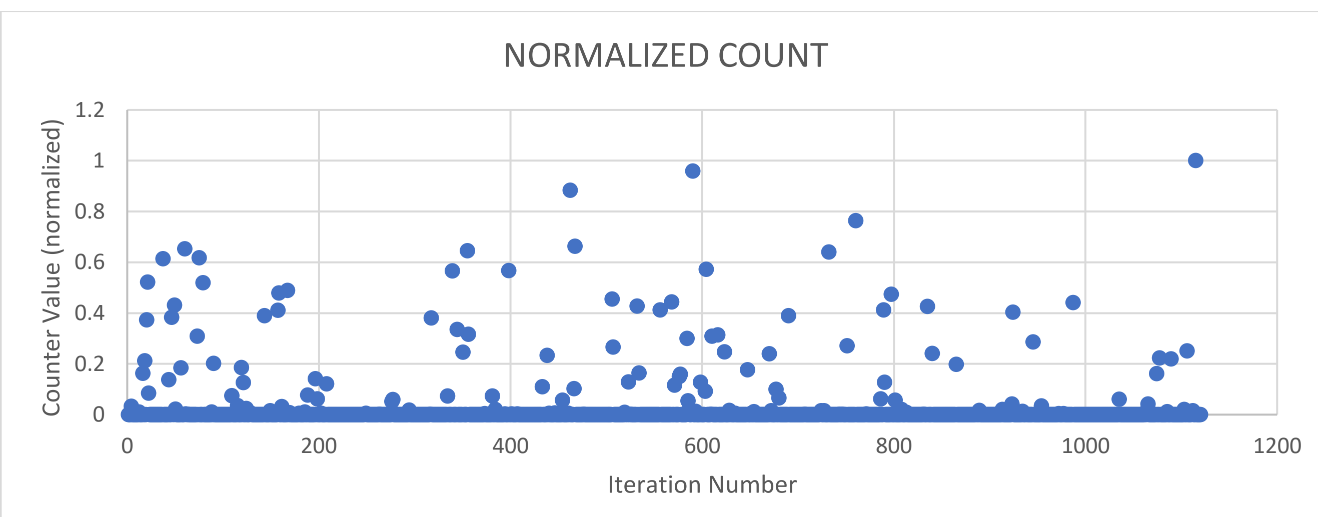

Figure 46: Scatterplot of outputs of a Neural Netowrk with LLC victims, normalized.

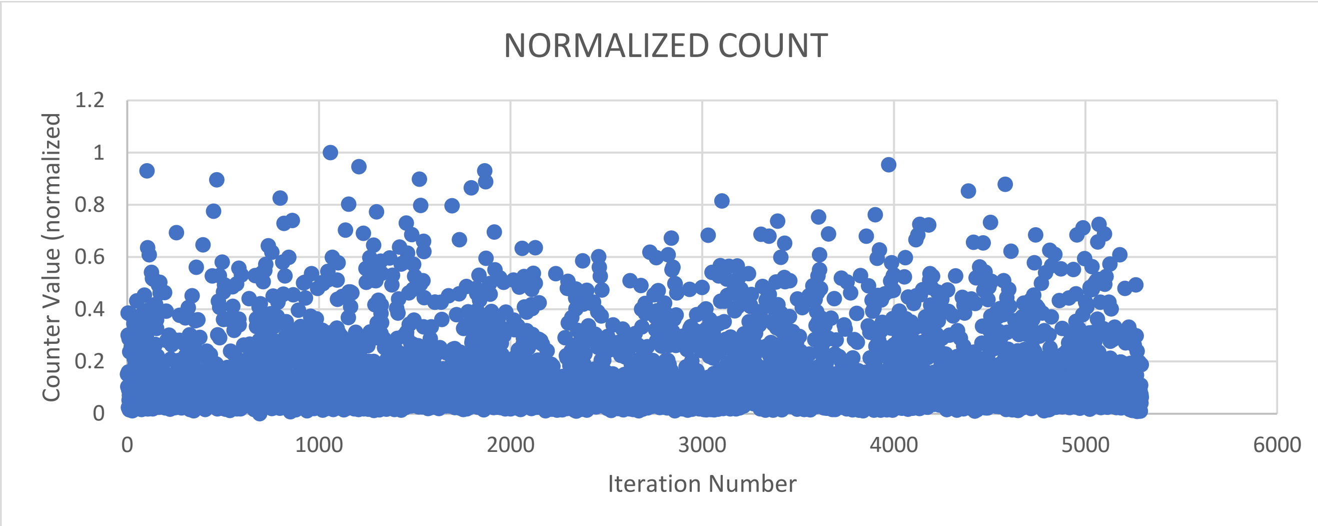

Figure 47: Scatterplot of outputs of a Neural Netowrk with Internal PMON 1, normalized.

We can see that for all three scenarios the behavior is quite inconsistent and erratic, without converging at any point or maximum. For figure 46, the algorithm struggled heavily during the entirety of the test to move the counter at all. While the other two were much better, the output was very inconsistent and dropped from the previous maximum (which is a very undesirable behavior).

In the current state of the algorithm, the output performance may just be as good as just letting the system run at random without any reinforcement in which case it would be more desirable than using a neural network. As such, with all the restrictions and disappointing performance, Neural Network will not be considered as an algorithm to include in the overlay graphs and comparison analysis.

Chapter 5: Analysis and Conclusions

5.1 Algorithm comparisons

With all the data compiled, it can be overlayed for comparison, to have a great way of visualizing the performance of all algorithms compared to each other. For this purpose, two types of graphs were made: one for the averages and one for maximums (one of each type per PMON). Scatterplot graphs were excluded in this type of analysis.

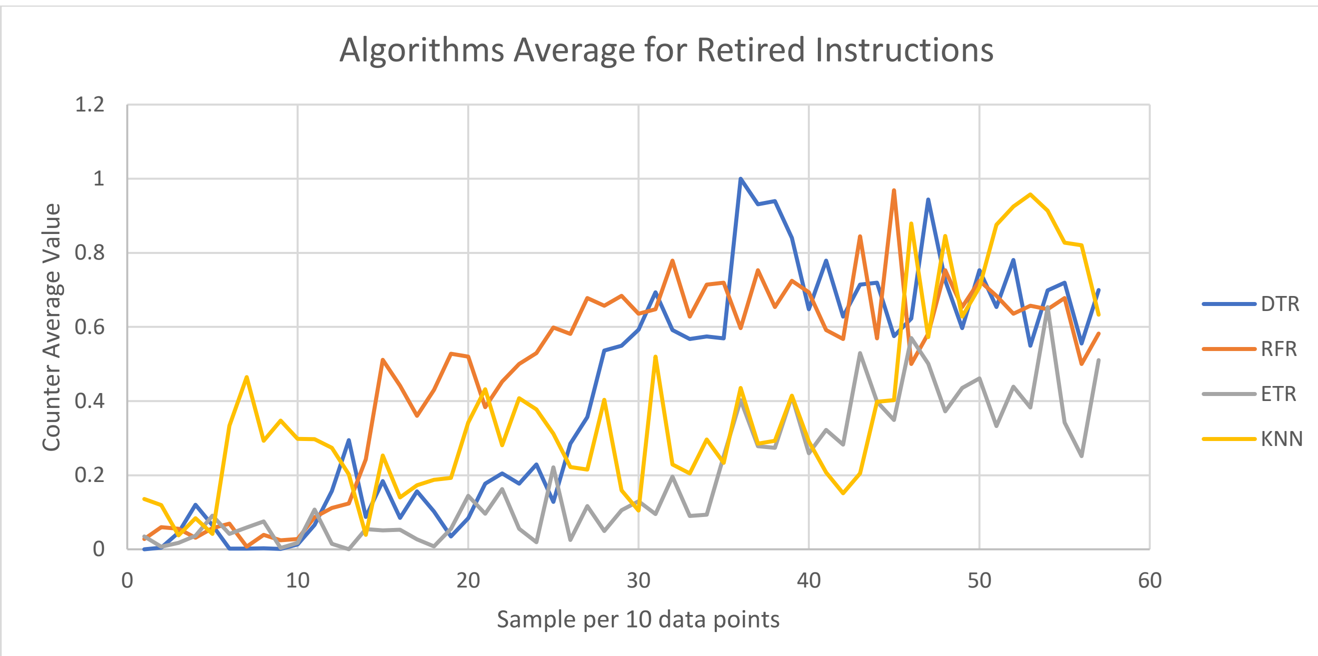

Figure 48: Averages of outputs for all algorithms overlapped, for Retired Instructions.

For retired instructions in figure 48, we can see KNN (yellow) was the first one to show signs of convergence, considerably sooner than all the other algorithms, but did so at a relatively low value of 0.3. While Random Forest (orange) took some iterations more, the early convergence was at a higher value around 0.4, but continued to climb to a much higher value, around 0.7. Decision Tree regressor (blue) took the longest to converge but did so at the second highest convergence value around 0.6 and achieved the highest average value of all algorithms of 1.

Additionally, KNN managed to 'jump' from its original convergence value to the highest of them all, at around 0.75, but took the longest to achieve its max value. Lastly, Extra Trees (gray) showed the worst performance overall, taking half the time to stabilize and had the lowest convergence value at 0.4.

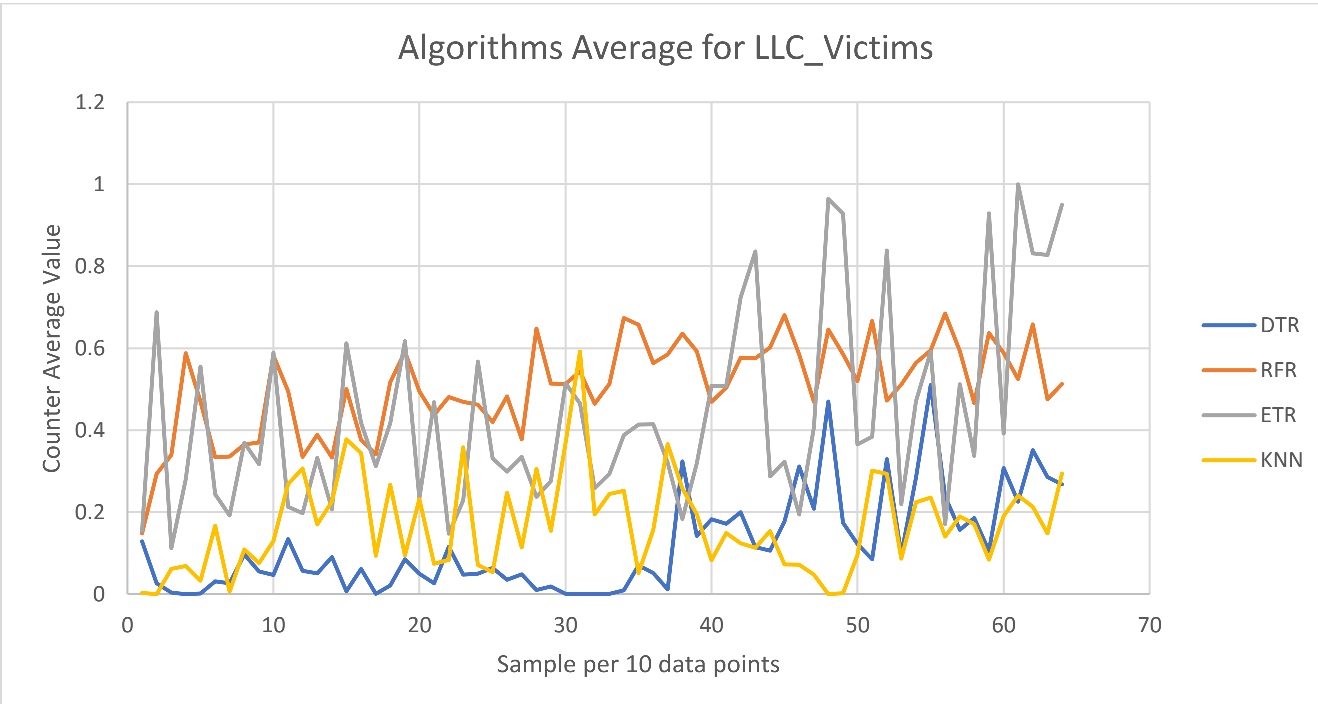

Figure 49: Averages of outputs for all algorithms overlapped, for LLC Victims.

Analyzing LLC victims in figure 49, the algorithm's behavior also changes drastically. Extra Trees model (gray) is now the algorithm that managed to hit the highest values throughout the entire test, and Decision Tree (blue) is in average the worst. KNN (yellow) is now much more erratic and noisier and fails to converge to a higher value after hitting a very low line around 0.2.

While Extra Trees (gray) was the fastest to reach some point of convergence, Random Forest (orange) follows almost immediately after but continues to slowly rise its convergence point to be the highest in average, making it the best to maintain stress in this algorithm. Additionally, Extra Trees model (gray) still manages to find the absolute maximum values at multiple points considerably higher than all the other algorithms.

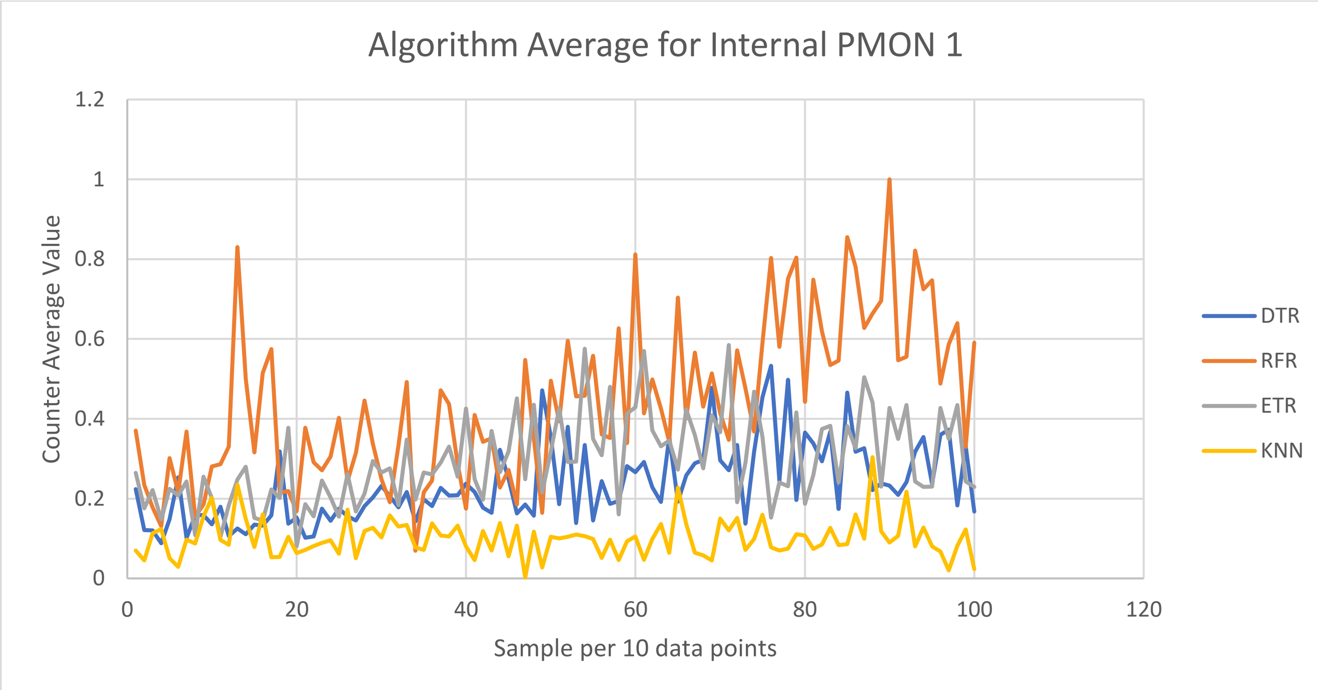

Figure 50: Averages of outputs for all algorithms overlapped, for Internal PMON 1.

Evaluating the last counter (figure 50) we can see a change in behavior again. Random Forest (orange) is the fastest to converge and the finder of the highest overall counter values, as well as finding the highest converging line of all algorithms by a considerable margin. Extra Trees model (gray) follows closely the same behavior of Decision Tree but with more variation and slightly higher peaks and KNN is not the lowest performing algorithm in all aspects, failing to find a useful line of convergence.

Having analyzed averages, which represent the overall behavior of the algorithms, we cannot say that these graphs tell the full story, since the overall behavior is not the absolute metric. Overlay analysis and comparisons for maximum values were also made to observe the maximum values the algorithms managed to achieve with the different counters, which may also be a desired feature depending on the circumstances of the application.

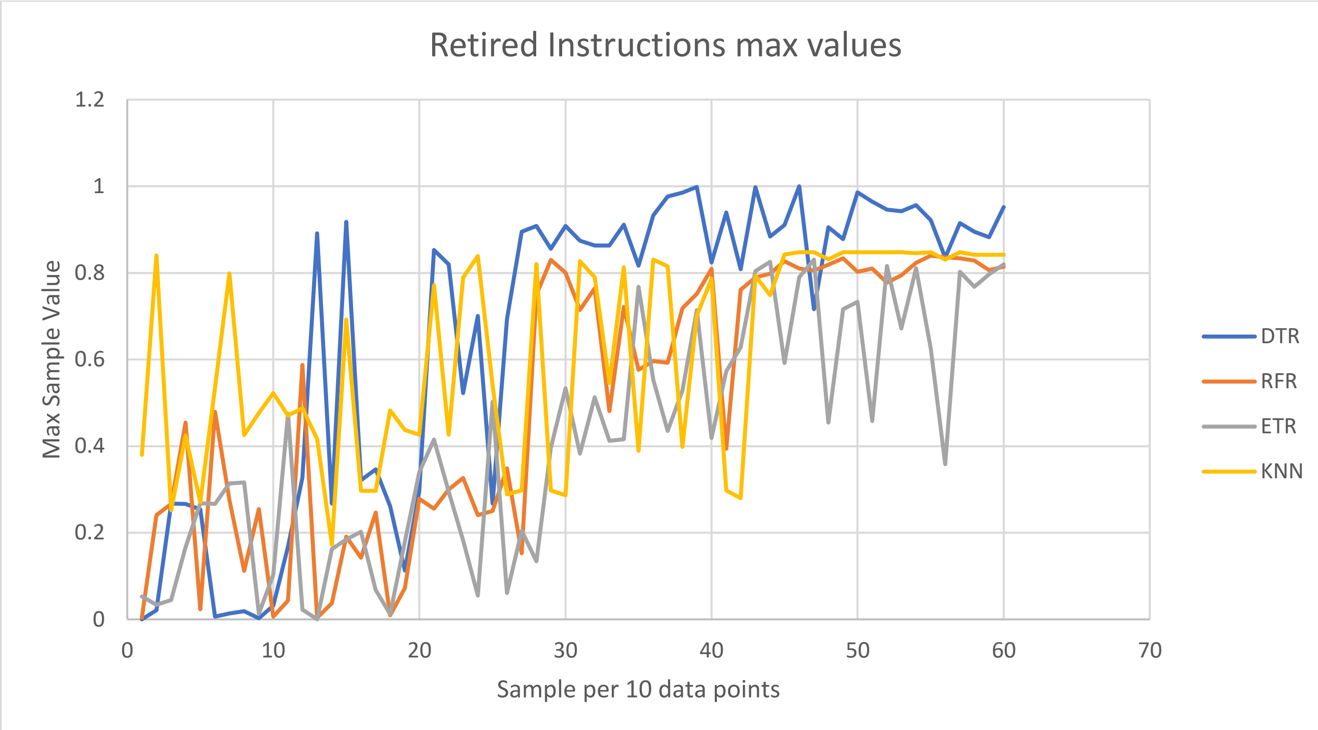

Figure 51: Maximum of outputs for all algorithms overlapped, for Retired Instructions.

Examining the first counter but looking at the maximums this time (figure 51) we can see some clear differences to analyze. Decision Tree (blue) managed to find the highest values of all algorithms and is also very stable, outpacing both Random Forest and Extra Trees in time to find those values as well.

KNN (yellow) shows a clear advantage reaching peak counter values almost from the start, and demonstrating one of the key behaviors of KNN: a flat line in the end (probably due to its tendency to 'snap' to a convergence line once it finds it). In this case the algorithm achieved its goal of maximizing the counter with very few repetitions elsewhere.

Random Forest (orange) took a little longer than Decision Tree (blue) to converge but managed to roughly match KNN (yellow) in counter value but falling behind in the early stages of the test. Extra Trees model (gray) is comparable to Random Forest in both iteration numbers and convergence line values, but has notably more variance throughout the test, without managing to achieve values higher than the rest.

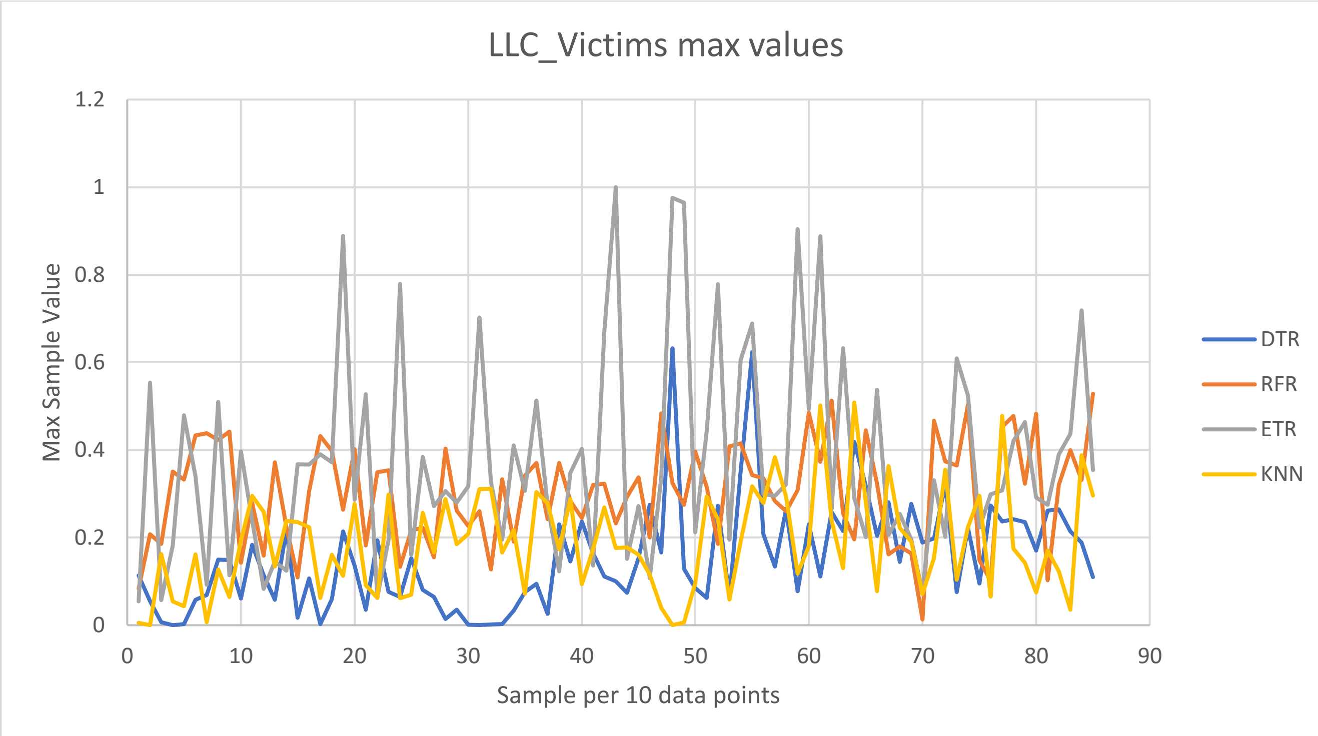

Figure 52: Maximum of outputs for all algorithms overlapped, for LLC victims.

Now seeing LLC victims counter (figure 52) we can see a similar behavior to the average overlay (figure 49) but much more disperse. Extra Trees model (gray) managed to find the highest value by a considerable margin, achieving almost double that of the other algorithms in some points but also having the most extreme variations. Decision Tree (blue) managed to get 2 samples with high values, making it the algorithm with the second highest values, noting that these are the only samples that it achieved, while having an overall average lower than the rest for most of the test.

Random Forest (orange) and KNN (yellow) had very similar results for this test, Random Forest being slightly higher overall throughout the entire test but both showing considerable variation and underwhelming values. Both do manage to have higher average values than Decision Tree but still failed to get the highest values of the group.

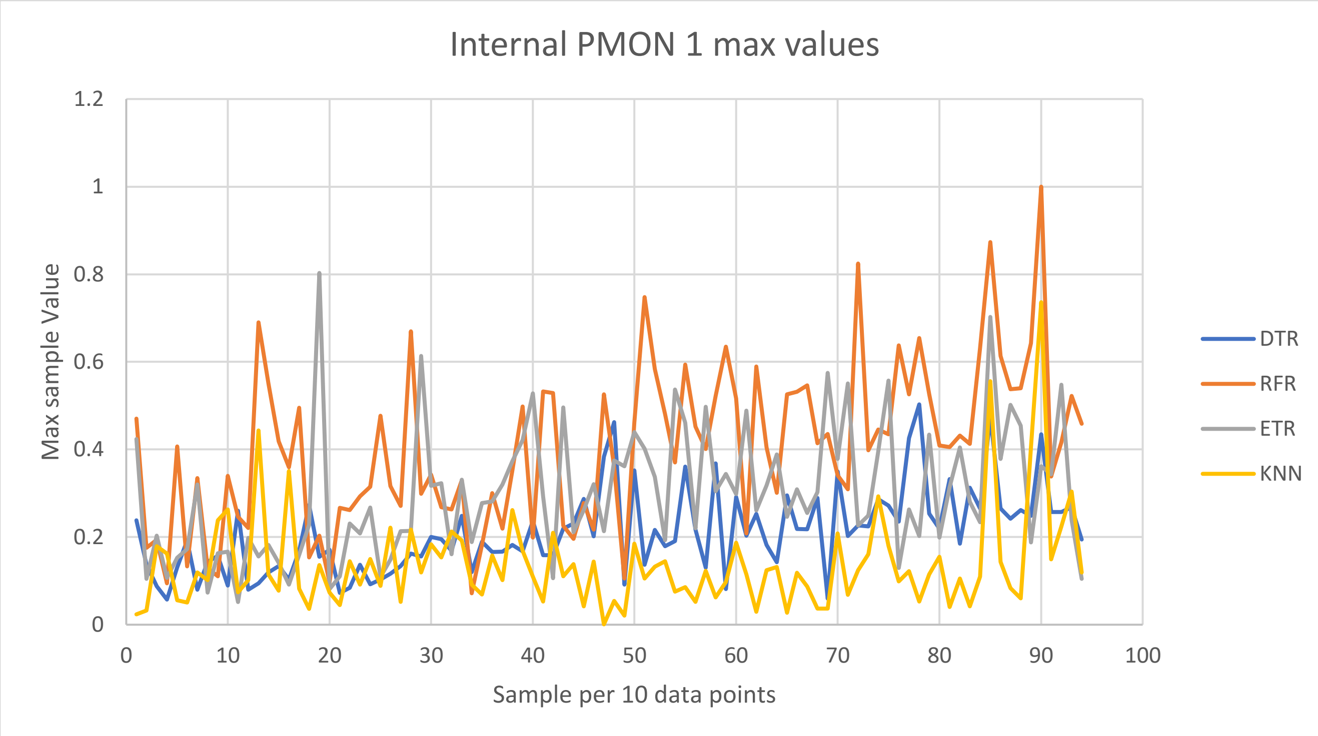

Figure 53: Maximum of outputs for all algorithms overlapped, for Internal PMON 1.

With the last test (figure 53) we can see the closest comparison in this document. Random Forest (orange) managed to get both the highest value of all algorithms and the highest average as well. However, it also shows the most variation (although followed closely by Extra Trees) and a tendency to go even higher if left to run for more iterations. Extra Trees model (gray) managed to get some peaks early in the test but failed to create higher values afterwards.

Decision Tree (blue) managed a decent average value (figure 50) but fell short in maximum values, although it was the algorithm with the least variation. Finally, KNN (yellow) kept showing low values during most of the test, except in the final part where we see part of KNN's key behavior in action, 'jumping' from the lowest convergence line of all algorithms to the second highest, managing to get the second highest value of all algorithms as well.

5.2 Algorithm Ranking

As explained in chapter 3, the performance results according to evaluation metrics are as shown in table 1 (1 means the best and 4 the worst).

		PERFORMANCE PLACING		
		Retired Instructions	LLC_victims	Internal Pmon 1
DTR	speed	3	4	3
	max avg	2	3	3
	variance	3	2	3
	abs max	1	2	4
RFR	**speed**	**2**	**2**	**1**
	max avg	**3**	**2**	**1**
	variance	**1**	**1**	**3**
	abs max	**3**	**3**	**1**
ETR	speed	4	1	2
	max avg	4	1	2
	variance	2	4	2
	abs max	4	1	2
KNN + DTR	speed	1	3	4
	max avg	1	4	4
	variance	4	3	1
	abs max	2	4	3

Table 1: Matrix of algorithm ranking results

Considering the rankings, now calculating the metric weights for each algorithm, shows the best overall algorithm, as shown in table 2.

		PERFORMANCE PLACING			
		Retired Instructions	LLC_victims	Internal Pmon 1	total
DTR	speed	1.33	1.00	1.33	
	max avg	1.50	1.00	1.00	
	divergence	0.67	1.00	0.67	
	abs max	1.00	0.50	0.25	
TOTAL		4.50	3.50	3.25	11.25
RFR	**speed**	**2.00**	**2.00**	**4.00**	
	max avg	**1.00**	**1.50**	**3.00**	
	divergence	**2.00**	**2.00**	**0.67**	
	abs max	**0.33**	**0.33**	**1.00**	
TOTAL		**5.33**	**5.83**	**8.67**	**19.83**
ETR	speed	1.00	4.00	2.00	
	max avg	0.75	3.00	1.50	
	divergence	1.00	0.50	1.00	
	abs max	0.25	1.00	0.50	
TOTAL		3.00	8.50	5.00	16.50
KNN + DTR	speed	4.00	1.33	1.00	
	max avg	3.00	0.75	0.75	
	divergence	0.50	0.67	2.00	
	abs max	0.50	0.25	0.33	
TOTAL		8.00	3.00	4.08	15.08

Table 2: Matrix of algorithm weighting results

Note: to see the formula for scoring calculation see chapter 3.2.

In summary, the ranking of algorithms as follows:

1. Random Forest Regressor
2. Extra Trees Regressor
3. KNN Regressor
4. Decision Tree Regressor

5.3 General Conclusions

As mentioned in chapter 3, while convergence speed is normally prioritized, it is not the only metric to consider. Absolute maximum value, average value of convergence and variance, among other minor behaviors, should be considered in the evaluation when choosing an algorithm for a particular application. For future applications, there may be completely different behaviors to be desired, such as (but not limited to) minimization, staying within a constrained window of values, replication of a particular value, etc.

It is worth mentioning that the implementations of the different optimizers for the algorithms may not be perfect nor the best versions that could be possibly used. Being the part responsible for manipulating the hyper-parameters, there may be better unexplored options not covered in this document.

As a general-purpose algorithm, for most tests, Random Forest can be safely chosen as the single best algorithm. Being the most consistent, converging faster than the others except in only one case and often getting the maximum (or close to) values as well. Additionally, it also has a low degree of variation. This means that Random Forest should be chosen for new tests or unexplored applications, since it has the most consistent qualities with little downsides.

Extra Trees model was the most disappointing algorithm, showing low convergence values and the most extreme variation in its outputs, while also taking a long time to find its convergence point. However, its extreme variation may also be its saving grace, since it was, without fail, the one that managed to find very high values in the shortest number of experiments, and also find the highest maximum values for some tests. This means that this algorithm may be an excellent choice if the intention is to prepare a 'random dataset' for use with other tests or algorithms.

Decision Tree demonstrates very inconsistent behavior, being the best in some aspects for some tests, and being the absolute worst for others. It managed to get the highest values for Retired Instructions, but the lowest for Intel PMON 1. It does not have as much variation as Extra Trees, however, so it also struggles to converge or find maximums early in the test. For some tests it does show promise of getting better with more iterations, and getting good stability once a convergence is found. This means that for long-term testing or for counters that are known to be 'easy to increase' this algorithm may show promise.

KNN mainly has a very good advantage and one big disadvantage, both derived from its ‘jumping behavior’. Once it finds a convergence line, it manages to stay in that line incredibly stable, but it also means that it often takes more time to ‘make the jump’. This behavior makes it exceptional for replicating maximum values it already found, making it a good choice for tests with the intention of maintaining a counter with a high value in constant stress, although the time it takes to converge also needs consideration.

Finally, Neural networks showed no semblance of convergence at all, with very extreme limitations, and failing to find higher values as well. This is probably due to how the implementation was done though, or maybe the way the reinforcement is made may not be the most adequate for this algorithm. Testing showed the training was not adequate and showed no evidence of any tendency.

For more conclusive testing (regarding Neural Networks), a new implementation would be needed, unconstrained by the limitations of the current one, such as reduced access to configuration options. Unfortunately, that exceeds the time available for this investigation.

All algorithms show different properties, some more beneficial than others, but it is ultimately the test that would define which one would be the best choice. It is still important to investigate the properties of each algorithm to make a proper choice so future investigations should be made with more, unexplored algorithms and maybe even a combination of different algorithms in different stages.

Appendix

All code snippets included were done by Santiago Soule exclusively. No code from Intel was included and some parts of Santiago's code may have been redacted to remove all references to Intel's code due to confidentiality reasons.

A.1 RANDOM FOREST AND EXTRA TREES OPTIMIZER CODE

```
import logging
import random
from typing import Callable
import numpy as np
from collections import Counter
import random
import TreeParams
import TreeAnalysis

logger = logging.getLogger(__name__)

class MultipleTreesAnalysis:
    single_tree_analyzer = TreeAnalysis()

    def get_model_analysis(self, model_name: str, params: dict) -> tuple:
        all_estimators_list = params['estimators']
        params.pop('estimators')
        test_suggest_values = {}
        ret = None
        dot_data = r""
        suggestions = []

        […]

        ###CONTRADICTION RESOLUTION AND MAX/MIN CALCULATION BELOW
        final_values = {}
        contradict_flag = False
        contradict_dict_a = {}

        for suggestion in suggestions:
            for key,value in suggestion.items():
                key_name_orig = key #extract string only
                key_name_opposite = ''
                less_flag = False
                span_flag = False
                if('<' in key_name_orig):
                   #create oposing key to check for contradictions later
                    key_name_opposite = key_name_orig.replace('<','')
                    less_flag = True
                else:
                    key_name_opposite = '<' + key_name_orig

                if((key_name_orig in final_values)
                and (key_name_opposite in final_values)):
                    span_flag = True
```

```
        else:
            span_flag = False


        if(key_name_orig in final_values): #check for conflicts
            if(less_flag): #less than
                print("conflict detected <<<")
                if(value < final_values[key_name_orig]):
                    if(span_flag):
                        if(value < final_values[key_name_opposite]):
                            #contradiction in a rage
                            contradict_flag = True
                            print("CONTRA IN range found in final_values")
                            contradict_dict_a.update({key_name_orig: value})
                            continue
                    else:
                        final_values[key_name_orig] = value
            else: #more than
                print("conflict detected >>>")
                if(value > final_values[key_name_orig]):
                    if(span_flag):
                        if(value > final_values[key_name_opposite]):
                            #contradiction in a range
                            contradict_flag = True
                            print("CONTRA IN range found in final_values")
                            contradict_dict_a.update({key_name_orig: value})
                            continue
                    else:
                        final_values[key_name_orig] = value


        elif((key_name_opposite in final_values) and less_flag):
            if(value > final_values[key_name_opposite]):
                #final is > but new_orig is <
                final_values.update({key_name_orig: value})
            else:
                contradict_flag = True
                print("CONTRA IN final is > but new_orig is <")
                contradict_dict_a.update({key_name_orig: value})

        elif(key_name_opposite in final_values): #potential contradiction
            if(value < final_values[key_name_opposite]):
                #new_orig is > but final is <
                final_values.update({key_name_orig: value})
            else:
                contradict_flag = True
                print("CONTRA IN new_orig is > but final is <")
                contradict_dict_a.update({key_name_orig: value})

        else: #if new suggestion not previously found
            final_values.update({key_name_orig: value})

print("final values dict: ")
print(final_values)

#Contradictions are when one tree says > than a value but another tree says
# < in a contradicting range. i.e. X = >2 && Y = <1.
```

```
 #Algorithm solves this via voting system (majority wins) or random when draw
if(contradict_flag):
    print("CONTRADICTION FLAG IS SET, resolving conflicts...")
    duplicates = []
    uniques = []

    for key,value in contradict_dict_a.items():
        if(key not in uniques):
            uniques.append(key)
        else:
            duplicates.append({key: values})

    contradictions = dict(Counter(duplicates))

    for key,value in contradictions.items():
        #since contradictions where found new suggestion will replace old one
        if('<' in key):#get oposing key for dicts
            opposite = key.replace('<','')
        else:
            opposite = '<'+key
        final_values.pop(opposite) #remove old entry
        final_values.update({key:contradict_dict_a[key]}) #add new entry

    for key in uniques:
        #since contradictions where only found ONCE for each node
        #value will be selected randomly
        rand = bool(random.getrandbits(1))
        if(rand): #if true then update for new suggestion, else use old one
            if('<' in key):#get oposing key for dicts
                opposite = key.replace('<','')
            else:
                opposite = '<'+key

            final_values.pop(opposite)
            final_values.update({key:contradict_dict_a[key]}) #add new entry

return ret, dot_data, final_values
```

A.2 KNN PREPROCESSOR CODE

```
import random
import os
import logging
import numpy as np
import pandas as pd
import math
import MLFrame
from sklearn.neighbors import NearestNeighbors
from difflib import get_close_matches

logger = logging.getLogger(__name__)

def search_for_cmd(cmd_2_search, df):
    m = cmd_2_search.iloc[0]["Index"].astype('int')
    if m > 0:
        a = df.iloc[m]
        return  f'Exact match: {a}'
    else:
        val =  get_close_matches(cmd_2_search, df.index)
        if len(val) > 0:
            return df.loc[val[0]]
        else:
            return "ERROR"

def predict_by_cmd(cmd, model, K):
    distance, suggestions = model.kneighbors(cmd.drop('test_cmd', axis=1), n_neighbors=K)
    return suggestions

def get_K_neighbors(cmd, df, K, model):
    search = search_for_cmd(cmd, df)
    if("Exact" in search ):
        neighbors = predict_by_cmd(cmd, model, K)
    elif("ERROR" not in search):
        neighbors = predict_by_cmd(search, model, K)
    else:
        neighbors = search
    return neighbors

def adapt_DF(knn_data, orig_df, inputs_df, event):
    reduced_data = orig_df.iloc[knn_data]
    reduced_data = reduced_data.drop('Index', axis=1)
    filtered_input_list = inputs_df.iloc[knn_data]
    core = event.replace(event.split('_')[-1],'')
    core = core[:len(core)-1]
    output_field = event.split('_')[-1]
    cmd_list = reduced_data["test_cmd"]
    wanted_output = reduced_data[event]
    reduced_data = reduced_data.drop('test_cmd', axis=1)
    return MLFrame(filtered_input_list, wanted_output, wanted_output, core, \
    output_field, None, cmd_list)

def KNN_analyze_and_reduce(data : MLFrame, event : str):
    orig_dataframe = data.dataframe
    sc_args_inputs_list = data.cmdlist
```

```
orig_dataframe = orig_dataframe.fillna(-1) #fill all NaN with -1 to avoid errors
num_of_cmds = len(orig_dataframe.index)
MAX_DATAPOINTS_OUTPUT = 1000
#min data points needs to match preproc_data_knn self._data or len(self._data)
#ni data points = minimum data points in general to start operating
#percentage total is the percentage of datapoints for KNN to output
#that will act as dataset for Decision Tree algorithm
MIN_DATAPOINTS_OUTPUT = 45
PERCENTAGE_OF_TOTAL = 0.35
number_of_neighbors = 5
if ((num_of_cmds * PERCENTAGE_OF_TOTAL) > MAX_DATAPOINTS_OUTPUT):
    number_of_neighbors = MAX_DATAPOINTS_OUTPUT
elif ((num_of_cmds * PERCENTAGE_OF_TOTAL) < MIN_DATAPOINTS_OUTPUT):
    number_of_neighbors = MIN_DATAPOINTS_OUTPUT
else:
    number_of_neighbors = math.ceil(num_of_cmds * PERCENTAGE_OF_TOTAL)
model = NearestNeighbors(metric="cosine", n_neighbors=number_of_neighbors)
model.fit(orig_dataframe.drop('test_cmd', axis=1))
max_desired_value_index = orig_dataframe[event].idxmax()
row_for_query = orig_dataframe.iloc[[max_desired_value_index]]
print("***************Found max value in DF: **************")
print(row_for_query)
print("proceeding to find similar neighbors through KNN query...")
new_data = get_K_neighbors(row_for_query, orig_dataframe, number_of_neighbors, model)
if(new_data is not "ERROR"):
    new_data = new_data.flatten()
else:
    print("!!!!!!!!!!!!ERROR, NEIGHBOR QUERY FAILED!!!!!!!!!!")
    return new_data
return adapt_DF(new_data, orig_dataframe, sc_args_inputs_list, event)
```

References

Amazon Web Services. (2024, January 6). *What is a Neural Network?* Retrieved from Amazon Web Services Learning Center: https://aws.amazon.com/what-is/neural-network/#:~:text=A%20neural%20network%20is%20a,that%20resembles%20the%20human%20brain

Amazon Web Services. (2024). *What is Reinforcement Learning*. Retrieved from AWS what is?: https://aws.amazon.com/what-is/reinforcement-learning/#:~:text=Reinforcement%20learning%20(RL)%20is%20a,use%20to%20achieve%20their%20goals.

Aziz, I. (2019, October 19). *Fundamental Knowledge of Machine Learning*. Retrieved from Medium.com: https://medium.com/analytics-vidhya/fundamental-omachine-learning-ada28afa1bd3

Barreto, A., Precup, D., & Pineau, J. (2014). Practical Kernel-Based Reinforcement Learning. *arXiv*, 60.

Brown, S. (2021, April 21). *Machine Learning, Explained.* Retrieved from MIT ideas made to matter: https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained

Cambridge. (2024, Febuary 21). *Cambridge Dictionary*. Retrieved from Benchmark definition: https://dictionary.cambridge.org/dictionary/english/benchmarking

Carden, S. W. (2014). Convergence of a Q-learning Variant for Continuous States and Actions. *Journal of Artificial Intelligence Research*, 28.

Carden, S. W. (2017). Small-sample reinforcement learning: Improving policies using synthetic data. *Intelligent Desicion Technologies*, 31.

Delua, J. (2021, march 12). *Supervised versus unsupervised learning: What's the difference?* Retrieved from IBM think: https://www.ibm.com/think/topics/supervised-vs-unsupervised-learning

Drucker, H. (1997). Improving Regressors using Boosting Techniques. *14th International Conference on Machine Learning of Monmouth University*, 9.

Ernst, D., Geurts, P., & Wehenkel, L. (2005). Tree-Based Batch Mode Reinforcement Learning. *Journal of Machine Learning Research*, 54.

Foote, K. D. (2021, December 3). *A brief History of Machine Learning*. Retrieved from datadiversity.net: https://www.dataversity.net/a-brief-history-of-machine-learning/

Géron, A. (2019). Decision Trees. In A. Géron, *Hands-on Machine Learning with Scikit-Learn, Keras & Tensorflow* (p. 50). Sebastopol: O'Reilly Media.

Géron, A. (2019). *Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow.* Sebastopol: O'Reilly Media.

IBM. (2024, January 9). *AI governance for the enterprise*. Retrieved from IBM think: https://www.ibm.com/topics/neural-networks

IBM. (2024, January 9). *What is a Decision Tree?* Retrieved from IBM think: https://www.ibm.com/topics/decision-trees#:~:text=A%20decision%20tree%20is%20a,internal%20nodes%20and%20leaf%20nodes

Intel. (2024, January 26). *Intel Perf Mon*. Retrieved from intel perfmon reference: https://perfmon-events.intel.com/

Intel. (2024, June). *Intel software development manual.* Retrieved from Intel architecture development manuals: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Jiang, N., & Singh, S. (2015). Abstraction Selection in Model-Based Reinforcement Learning. *Michigan University Machine Learning Innovation Conference*, 14.

Klokova, M. (2023, September 23). *Important dates in history of Machine Learning*. Retrieved from Medium.com: https://medium.com/@marizombie/important-dates-in-history-of-machine-learning-d52d7f24b797

Luca Invernizzi, J. L. (2021, October 27). *Getting started with KerasTuner*. Retrieved from Keras User Guides: https://keras.io/guides/keras_tuner/getting_started/

NumFocus Inc. (2024, September 20). *Pandas Documentation*. Retrieved from Pandas User Guide: https://pandas.pydata.org/docs/user_guide/index.html#user-guide

Numpy Developers. (2024, March 2). *Numpy user Manual*. Retrieved from NumPy Documentation: https://numpy.org/doc/2.1/

Our world in Data. (2023, September 18). *Charted: The Exponential Growth in AI Computation*. Retrieved from visual capitalist: https://www.visualcapitalist.com/cp/charted-history-exponential-growth-in-ai-computation/

Rogers, A. (2019, May 9). *What Deep Blue And AlphaGo Can Teach Us About Explainable AI*. Retrieved from Forbes: https://www.forbes.com/sites/forbestechcouncil/2019/05/09/what-deep-blue-and-alphago-can-teach-us-about-explainable-ai/?sh=24337eeb52fd

Schnitzler, F., Wehenkel, L., & Ernst, D. (2011). Optimal Sample Selection for Batch-mode Reinforcement Learning. *University of Liège Innovation Conference*, 10.

Scikit Learn. (2024, January 3). *Clustering*. Retrieved from Scikit Learn documentation: https://scikit-learn.org/stable/modules/clustering.html

Scikit Learn. (2024, March 15). *Decision Trees Documentation*. Retrieved from Scikit Learn model documentation: https://scikit-learn.org/stable/modules/tree.html

Scikit-Learn. (2024, March 4). *KNN Nearest Neightbors*. Retrieved from scikit learn user guide: https://scikit-learn.org/dev/modules/neighbors.html#neighbors

Todorov, N. (2014, june 14). *A modern smartphone or a vintage supercomputer: which is more powerful?* Retrieved from phone arena news: https://www.phonearena.com/news/A-modern-smartphone-or-a-vintage-supercomputer-which-is-more-powerful_id57149

Trevor Hastie, R. T. (2008). *The Elements of Statistical Learning.* Stanford, California: Springer.

Udluft, S., & Hans, A. (2009). Efficient Uncertainty Propagation for Reinforcement Learning with Limited. *Siemens conference of AI innovation*, 11.