Will It Rain Tomorrow?

Parallel Hyperparameter Tuning for Machine Learning Algorithms

Group 9
Abhinav Behal, Matthew Frost, Molly Farrant
abeh957, mfro529, mfar672
GitHub Repository: https://github.com/AbhinavBehal/751-project

Introduction

Machine learning (ML) is a popular branch of artificial intelligence (AI) that essentially aims to derive predictive models from large datasets [1]. ML algorithms require increasingly large numbers of hyperparameters that influence the performance of the models they produce. Finding optimal combinations of such parameters through hyperparameter tuning is computationally expensive and time-consuming. However, an increase in the prevalence of cloud computing in recent years has afforded practitioners easy access to parallel and distributed computing resources [2]. As a result of this, horizontally scaling the hyperparameter tuning process is now both feasible and desirable. Optimising computationally intensive models through massive parallelism is arguably the new paradigm for hyperparameter tuning. This project aims to implement an effective hyperparameter tuning algorithm for machine learning models that leverages parallel and distributed computing techniques. It is hoped that a quality predictive model, able to accurately predict the advent of rain in Australia, can be derived from the Kaggle dataset "The Rain in Australia" [3]. Ultimately, this project aims to answer the following research questions:

Research Question 1: "To what extent does hyperparameter tuning affect the quality of predictive models derived from 'The Rain in Australia' dataset?"

Research Question 2: "What impact does parallelism have on the performance of hyperparameter tuning algorithms when applied to 'The Rain in Australia' dataset?"

1. Fundamentals of Machine Learning

Machine learning (ML) is a subset of artificial intelligence (AI) that combines the disciplines of computer science and statistics. It facilitates automatic learning and improvement through experience without the need for explicit programming [1]. AI practitioners have deduced that exposing programs to input-output behaviour and allowing them to learn autonomously is more effective than explicitly programming desired behaviour for all possible inputs. This knowledge, paired with the increasing availability of big data and low-cost computation, is fuelling rapid growth of the ML field [2].

1.1 Techniques and Trends

ML techniques can be classified into three distinct categories: reinforcement learning, unsupervised learning and supervised learning [1]. In reinforcement learning, the learner receives rewards and punishments for specific actions. The algorithm leverages this feedback to maximise rewards, minimise punishments, and ultimately exact ideal behaviour [2]. Unsupervised learning involves the analysis of unlabelled data with the aim of identifying prevalent patterns or trends. In contrast, supervised learning involves labelled datasets with training examples. Regressions are examples of supervised learning techniques in which the mathematical relationship between two or more variables is analysed and quantified. Classification algorithms also involve supervised learning and are used to classify data into predefined categories [1]. A classification problem is deemed binary when data is simply classified into two distinct categories.

Ensembles are popular ML algorithms that combine several base models to produce one aggregate predictive model [1]. Regression and classification problems can both be solved through the application of an ensemble. There are two main families of ensemble algorithm: bagging and boosting. In bagging,

short for bootstrap aggregating, each model is exposed to a unique subset of the training data. Models are built independently and are subsequently combined via averaging or majority voting [1]. Boosting, however, is a sequential approach. Each model is exposed to the entire training dataset, and the models are built in succession. Each new model is influenced by the performance of those built previously [1]. Gradient boosting is a type of boosting algorithm in which each new model produced aims to minimise a given loss function. A loss function is a measure of how well a predictive model fits the relevant training data. The more accurate the model, the lower the loss function.

1.2 ML and Parallelism

Parallelism can be applied to several aspects of ML: the algorithm itself, training, and validation. Existing literature discusses the parallelisation of bagging algorithms. Because bagging involves training models independently of one another, these algorithms are embarrassingly parallel; dividing them into smaller subproblems for the sake of parallelisation is trivial [2].

The sequential nature of boosting renders the training process more difficult to parallelise. However, *XGBoost* is an implementation of gradient boosting designed for speed and performance. It is a library for developing fast and high-performance gradient boosting tree models [4]. *XGBoost* provides the parallelisation of tree construction using all the available CPU cores during training. Models are still built sequentially, but the construction of a single tree is carried out on multiple cores – a form of low-level parallelism. *XGBoost* also offers distributed computing for training very large models using a cluster of machines. Finally, *XGBoost* provides cache optimisation of data structures and algorithms to make the best use of available hardware [4].

Building a predictive model using *XGBoost* is straightforward. However, its numerous parameters mean that improving a model's performance is challenging. Selecting a set of parameters to tune and determining their optimal values is known as hyperparameter tuning [5]. *Google Vizier* is a black-box optimisation service that is said to have become the "de facto parameter tuning engine at Google" [6]. Optimising the process of hyperparameter tuning for the *XGBoost* implementation of gradient boosting is the focus of this project and is discussed in depth in subsequent sections.

2. Hyperparameter Tuning

In the context of ML, a hyperparameter is an algorithm parameter set prior to the training process [5]. ML algorithms typically have a dozen or more different hyperparameters that influence the performance of the resulting model (i.e. accuracy, training time, etc.) Due to the sheer number of possible hyperparameter configurations, evaluating each one is completely infeasible. The following algorithms can be used to carry out hyperparameter tuning for ML algorithms.

2.1 Grid Search

Grid Search is a prevalent approach to hyperparameter tuning. Its simplicity and ease of implementation have made it ubiquitous in popular machine learning packages, such as *scikit-learn*. The Grid Search algorithm exhaustively searches a subset of the hyperparameter space to find the optimal configuration based on a predefined performance metric. The independent nature of the configurations allows them to be trivially evaluated in parallel. However, due to a large number of hyperparameters, the search must be relatively coarse to avoid an exponential increase in the number of combinations. Furthermore, exacting a set of possible values for each hyperparameter is cumbersome, thus fine-tuning Grid Search can be difficult. Overall, the coarseness of the search and the manual specification of possible hyperparameter values typically renders Grid Search performance poor. However, its simple nature makes it suitable for ML algorithms with few hyperparameters, or for quick prototyping.

2.2 Random Search

Random Search is another popular hyperparameter tuning approach found in most popular ML packages. The algorithm begins by selecting a hyperparameter configuration at random from a user-defined set of hyperparameter distributions. It then evaluates the selected configuration based on some performance metric, and the process is repeated. The algorithm terminates upon reaching a pre-defined number of iterations. Similarly to Grid Search, each configuration can be evaluated independently,

making the algorithm trivially parallelizable. The random nature of the algorithm makes it much simpler to use than Grid Search, as hyperparameter distributions are easier to specify than possible hyperparameter values. Additionally, the exponential increase in hyperparameter combinations is avoided without having to make the search more coarse-grained. However, the random nature of the algorithm also means that the configurations aren't chosen in any intelligent way. Though a large number of iterations increases the probability that the resulting configuration will be optimal, this is not guaranteed.

2.3 Successive Halving Algorithm (SHA)

Compared to Grid and Random Search, SHA is complex. However, its approach is still relatively straightforward. The algorithm works by randomly generating N different configurations, assigning each of them a minimum number (Min_R) of epochs, such as iterations or time to run. The configurations are then evaluated, with the top performing 1/RF% being promoted to the next "rung" and given RF times the number of epochs (RF = Reduction Factor). This proceeds until the final rung is reached, meaning configurations have a maximum number (Max_R) of epochs. The best configuration is then selected from the final rung [7]. SHA can be thought of as a more intelligent Random Search, as only the best performing configurations are given more time to execute, allowing for more efficient resource allocation. However, this greediness may bias results, as certain configurations could perform better than others if given more time to run. The synchronous nature of SHA, specifically the fact that all configurations on the current rung must be evaluated before promotion takes place, makes parallelisation attempts relatively ineffective. Furthermore, the number of configurations reduces as the algorithm proceeds, thus so do the number of configurations that can be evaluated in parallel. As a result, the highest rungs utilise few processing cores.

2.4 Asynchronous Successive Halving Algorithm (ASHA)

ASHA is an extension of SHA that utilises asynchronous functions to afford parallelism. At its core, the algorithm is very similar to SHA, with the only difference being the promotion scheme used. ASHA promotes configurations to the next rung when the number of configurations in the rung below exceed RF. If there are no promotable configurations, a new random configuration is generated, growing the bottom-most rung [7]. This promotion scheme allows for a greater degree of parallelism, with configurations on different rungs being evaluated simultaneously, unlike SHA. Furthermore, this promotion scheme also allows for a greater number of configurations to be evaluated in the same amount of time. However, the promotion scheme could result in the sub-optimal promotion of some configurations that would not have been promoted in SHA. Additionally, as with SHA, the introduction of several meta-parameters for the tuning algorithm (e.g. Min_R, Max_R, RF) make this algorithm complex and difficult to fine-tune.

3. K-fold Cross-validation

Cross-validation is a technique used to assess the performance of machine learning models. It evaluates the performance of a given model with respect to some metric and is useful for determining how well the model will perform in practice [8]. In general, it involves splitting a dataset into distinct training and validation sets, exposing models to the training set, and checking their performance against the validation set. Cross-validation can be useful for determining an algorithm's optimal set of hyperparameters, as these values will result in the lowest test error. K-fold cross-validation is a cross-validation method that reduces underfitting and overfitting, and 3-fold cross-validation was utilised in this project to validate the hyperparameter configurations produced by each of the four algorithms under test.

4. Dataset

The dataset used for this project was "The Rain in Australia" dataset sourced from Kaggle [3]. This is a binary classification dataset used to predict whether it will rain tomorrow in Australia. It contains 142,193 data points and 24 meteorological features, such as MinTemp, MaxTemp and Rainfall. Before the dataset was analysed, pre-processing was required to improve its quality and correctly format it for classification.

First, features columns that were not useful were dropped, as they contained too many null values or leaked too much information to the classifier. For example, the "RISK_MM" was directly used by the creators of the dataset to determine the RainTomorrow column. Subsequently, rows that contained null values or numerical outliers, data points more than 3 standard deviations away from the mean, were dropped. This action reduced the number of data points by 34,325 to 107,868. The data was then correctly formatted using binary encoding for the RainToday and RainTomorrow columns, and one-hot encoding for the categorical columns of wind direction. Finally, the data was standardised using min-max scaling: a technique used to normalise the range of independent variables [9].

5. Implementation

Several libraries developed for *Python* were used to facilitate our implementation: *Pandas*, *NumPy*, *SciPy*, *Scikit-Learn* and *XGBoost*. Pandas is an open-source library providing high-performance data analysis tools. *NumPy* and *SciPy* were used in conjunction with Pandas for data handling and analysis purposes. Machine learning models were built with the aid of the *Scikit-Learn* library and the *XGBoost* library was, of course, used to execute the *XGBoost* algorithm on the chosen dataset.

Grid Search and Random Search were implemented using the *Scikit-Learn* ML package directly. While SHA and ASHA were implemented from scratch using the pseudo-code provided in [7]. Grid Search, Random Search and SHA all use process-based parallelism to evaluate configurations in parallel, whereas ASHA was implemented using an AWS Lambda function, allowing for massive parallelisation with upwards of 500 workers evaluating configurations at the same time. AWS Lambda was chosen for ASHA due to its ability to automatically scale regardless of computational load [10]. ASHA uses a *ThreadPoolExecutor* (from the python standard library) to make asynchronous calls to the Lambda function. Each invocation of the Lambda function was placed on its own thread since the configurations are not being evaluated on the main machine. The main thread then polls this list of workers, and as soon as a worker finishes evaluating its current configuration, it is assigned a new configuration immediately to minimise worker idle time and maximise the total number of configurations evaluated. Evaluated configurations are kept in a priority queue so that the best performing configurations can be retrieved with little runtime overhead.

The different tuning algorithms are invoked through a command-line interface, and users are able to specify the algorithm they want, along with the algorithm-specific parameters to use, e.g. Min_R for ASHA. All algorithms are able to run in both sequential and parallel mode, and this can be specified through the n_workers input parameter. Usage details for all four tuning algorithms can be found in the GitHub repository readme file.

6. Results and Discussion

All benchmarks utilised 3-fold cross-validation and were executed on a laptop with 8GB RAM and an Intel Core i7-6500U CPU (2 physical cores that are Hyperthreaded to 4). Grid Search, Random Search, SHA and ASHA were applied to the dataset using bash scripts and their relative performance was evaluated with respect to two metrics: runtime and accuracy. Grid Search and Random Search were both executed using 4 workers, SHA was executed both sequentially (1 worker) and in parallel (4 workers), and ASHA was executed using 50, 200, and 500 workers. Additionally, variations of ASHA were tested and compared. Raw data from the executed benchmarks can be found in the *results.xlsx* file in the GitHub repository.

6.1 Relative Algorithm Performance

For each of the hyperparameter tuning algorithms evaluated, Figure 1. illustrates the relationship between runtime and accuracy of the resulting models, while Figure 2. demonstrates the relationship between runtime and number of configurations evaluated.

Random Search consistently produced the least accurate models. A single iteration produced a model with 81.89% accuracy in 6.18 seconds. Executing 256 iterations took 646.11 and resulted in 84.16%

accuracy. On average, accuracy increased at a rate of 0.02% per second of execution. Grid Search also performed relatively poorly. Evaluating a single configuration took 5.20 seconds and produced a model with 83.72% accuracy. After 909.12 seconds, 300 configurations have been evaluated and accuracy has increased to 84.25%. Using this algorithm, accuracy only increased at an average rate of 0.0006% per second. Additionally, the maximum accuracy produced by Grid Search was only 0.09% higher than that of Random Search but took 263.01 seconds longer.

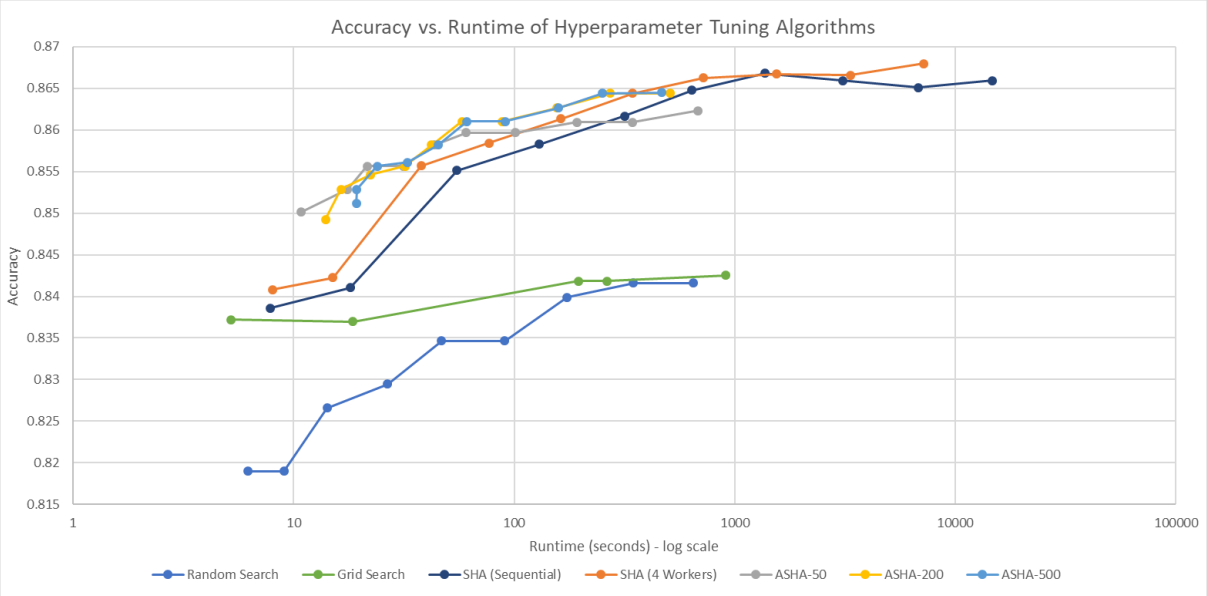

Figure 1.

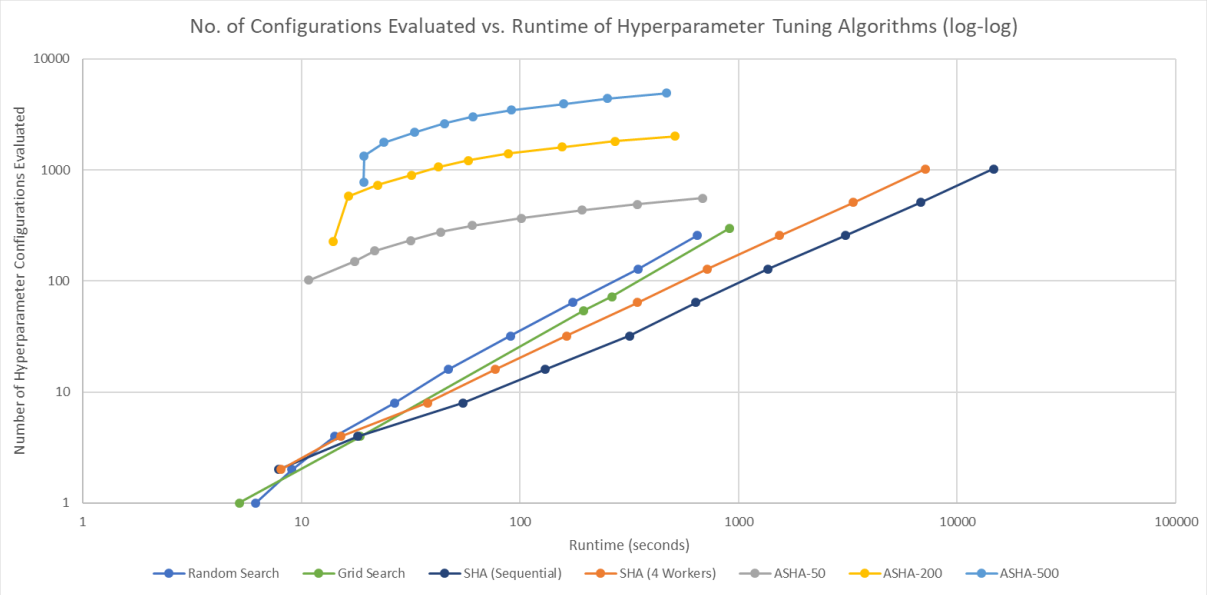

Figure 2.

SHA with 4 worker threads significantly outperformed both Grid Search and Random Search with respect to time and accuracy. It was able to evaluate two configurations in 8.05 seconds and the resulting model had an accuracy of 84.08%. After 37.77 seconds, the algorithm evaluated 8 configurations and accuracy increased to 85.57%. This model was 1.41% more accurate than the most accurate model produced by Random Search, and 1.32% more accurate than that of Grid Search, in approximately one-twentieth of the time. The evaluation of 1024 configurations under SHA took 7173.3 seconds and produced a model with 86.80% accuracy, 2.55% higher than the most accurate model produced by Grid and Random Search.

ASHA was evaluated with 50 (ASHA-50), 200 (ASHA-200) and 500 (ASHA-500) worker threads. ASHA-50 was able to evaluate 102 configurations in 10.80 seconds, resulting in accuracy of 85.01%, and 555 configurations in 683.45 seconds, producing an accuracy of 86.23%. ASHA-200 evaluated 228 configurations in 13.97 seconds and produced an accuracy of 84.93%. After evaluating 2015 configurations in 510.84 seconds, the model accuracy increased to 86.44%. ASHA-500 took 19.27 seconds to evaluate 770 configurations, resulting in 85.12% accuracy, and 467.08 seconds to evaluate 4901 configurations, producing 86.45% accuracy. ASHA-500 thus produced a marginally more accurate model than ASHA-50 and ASHA-200 in a shorter period of time. To achieve such accuracy, it evaluated 8.8 times as many configurations as ASHA-50, and 2.4 times as many as ASHA-200. When comparing SHA and ASHA in parallel, a trade-off between speed and accuracy is evident.

To produce an accuracy of 86.44%, SHA with 4 workers took 344.11 seconds while ASHA-500 took just 250.57 seconds; ASHA-500 provided an approximate 1.35 speedup over SHA in parallel. However, SHA was eventually able to produce a model 0.4% more accurate than the best effort of any ASHA variation. A baseline sequential version of SHA was also executed and took 638.44 seconds to produce the accuracy achieved by ASHA. ASHA thus gave an approximate speedup of 2.55 over the sequential SHA algorithm. Again, SHA was able to achieve marginally higher accuracy, 0.14%, but took significantly longer than ASHA to do so. ASHA clearly exhibits a desirable trade-off between runtime and accuracy when compared to sequential and parallel versions of SHA.

6.2 Performance of ASHA Variations

The performance of an ASHA implementation with 100 worker threads (ASHA-100) was evaluated with respect to several variables. The algorithm's Minimum R, Maximum R, Reduction Factor, and Early Stopping Rounds parameters were adjusted and their respective effects on accuracy and runtime were measured.

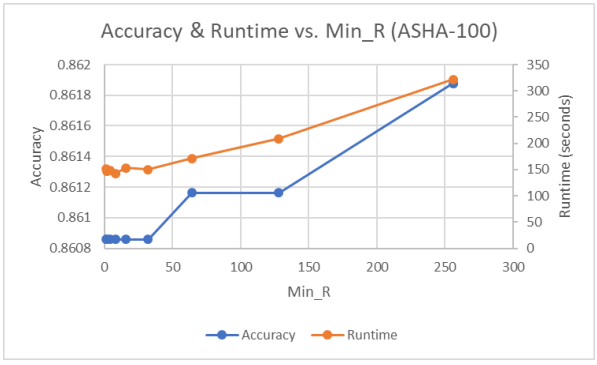

Figure 3.

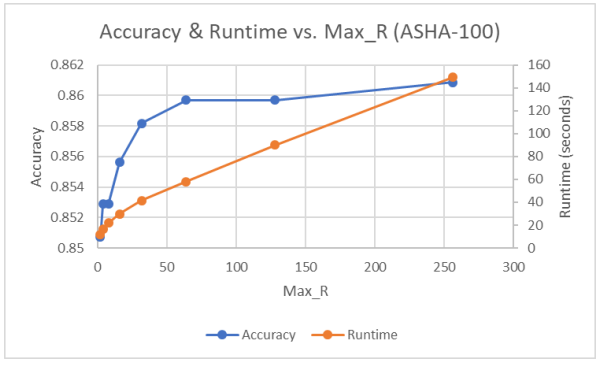

Figure 4.

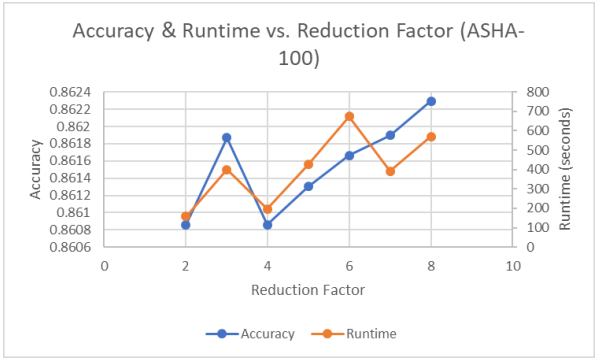

Figure 5.

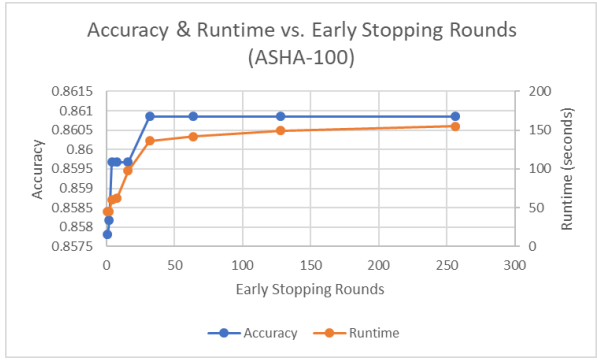

Figure 6.

Figure 3. illustrates the effect of the Minimum R value on the accuracy and runtime of ASHA-100. As the value of Minimum R increased, so did accuracy and runtime; positive, and somewhat linear correlations between both accuracy and runtime and the value of Minimum R were evident. Increasing the Minimum R from 1 to 256 caused runtime to increase by 170.63 seconds and resulted in a small

0.10% increase in accuracy. From this, we can infer that the small increase in accuracy due to large values of Minimum R do not outweigh their significant negative effect on runtime.

The effects of Maximum R on the accuracy and runtime of ASHA-100 is demonstrated in Figure 4. Again, a positive, linear correlation was identified between the value of Maximum R and runtime; as Maximum R increased, so did runtime. For low values of Maximum R between 2 and 32, accuracy increased 0.75%. However, as Maximum R was increased further, the increase in accuracy began to taper off. Between the values of 64 and 256, accuracy increased by only 0.12%. The effect on accuracy is thus only significant for relatively low values of Maximum R. Maximum R values in the range of 32 to 64 thus offer the best trade-off between runtime and accuracy.

Changes in accuracy and runtime in response to the Reduction Factor value are shown in Figure 5. Above a Reduction Factor of 4, a positive, linear correlation was evident between Reduction Factor and accuracy. Additionally, there seemed to be a weak, positive correlation between Reduction Factor and runtime. The impact of this parameter on ASHA-100 was otherwise unclear, as runtime and accuracy measures fluctuated with changes in Reduction Factor. Figure 6. depicts the effects of the Early Stopping Rounds variable on ASHA-100. Between the values of 1 and 32, accuracy and runtime both increased significantly as Early Stopping Rounds was increased. However, both metrics were largely unaffected by the Early Stopping Rounds parameter above this value. Between the values of 32 and 256, runtime increased by just 18.77 seconds, while accuracy remained constant at 86.09%.

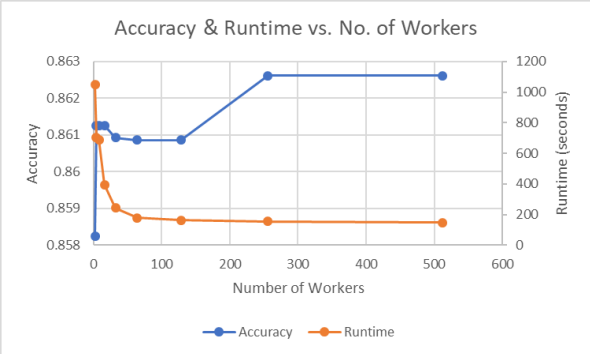

Figure 7.

The impact of the number of worker threads on ASHA was also investigated. The results are illustrated in Figure 7. As the number of workers was increased from 2 to 64, the algorithm's runtime exponentially decreased from 1050.59 to 146.96 seconds. The effect of additional workers on runtime decreased as more threads were added. The decrease in runtime between 128 and 512 threads was just 15.15 seconds. Similarly, increasing the number of threads from 2 to 4 resulted in a relatively large increase in accuracy of 0.3%. Interestingly, accuracy decreased following the introduction of more than 16 threads. Finally, for 256 worker threads or more, the algorithm's accuracy remained constant at 86.26%. Overall, increasing the number of threads only improved the accuracy of ASHA-100 by 0.44%, but significantly reduced the runtime by 903.63 seconds. A high number of worker threads is thus desirable due to its observed significant positive effects on performance.

6.3 Impact of Hyperparameter Tuning

XGBoost comes packaged with a default hyperparameter configuration. This configuration was used to analyse "The Rain in Spain" dataset and produce a predictive model. The model was derived in 4.06 seconds and gave an accuracy of 84.38%; a score only marginally better than what was achieved through Grid and Random Search. The default configuration accuracy was 2.42% lower than the maximum accuracy produced by SHA and 2.07% lower than that of ASHA. It is evident that hyperparameter tuning produces significantly more accurate predictive models when applied to ML algorithms, but the trade-off between accuracy and runtime needs to be considered.

Conclusion

In this project, ASHA, a hyperparameter tuning algorithm able to effectively leverage parallel and distributed computing techniques, was implemented. Overall, relatively more sophisticated hyperparameter tuning algorithms, SHA and ASHA overwhelmingly outperformed their less intelligent counterparts: Grid Search and Random Search. Additionally, the effects of parallelism on the process of hyperparameter tuning were found to be both positive and significant. ASHA was more efficient, but the resulting models were less accurate than those of SHA. ASHA also better utilised available processing units in the higher rungs of the algorithm. Strong evidence was presented for the fact that hyperparameter tuning does, in fact, cause machine learning models to produce higher-quality predictive models when applied to the dataset "The Rain in Australia".

Future work in this area could include running SHA via AWS Lambda and running ASHA locally to compare the results. Additionally, the promotion scheme of ASHA could be modified and the effects evaluated. The creation of reusable forms of SHA and ASHA, perhaps as an open-source library, would be beneficial for ML practitioners, as it would facilitate widespread use and execution of these algorithms on multiple datasets and in a myriad of environments.

References

[1] M. Heller, "Machine learning algorithms explained," *InfoWorld.Com*, 2019.
[2] M I Jordan and T M Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, *(6245)*, pp. 255-260, 2015. Available: https://www.ncbi.nlm.nih.gov/pubmed/26185243. DOI: 10.1126/science.aaa8415.
[3] J. Young. (January,). *Rain in Australia*. Available: https://kaggle.com/jsphyg/weather-dataset-rattle-package.
[4] J. Brownlee, "A Gentle Introduction to XGBoost for Applied Machine Learning," 2016. Available: https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/.
[5] T. Boyle. (-02-16T18:54:00.922Z). *Hyperparameter Tuning*. Available: https://towardsdatascience.com/hyperparameter-tuning-c5619e7e6624.
[6] D. Golovin *et al*, "Google vizier," in Aug 13, 2017, Available: http://dl.acm.org/citation.cfm?id=3098043. DOI: 10.1145/3097983.3098043.
[7] L. Li *et al*, "Massively Parallel Hyperparameter Tuning," 2018. Available: https://arxiv.org/abs/1810.05934.
[8] G. Drakos. (-08-16T21:59:10.938Z). *Cross-Validation*. Available: https://towardsdatascience.com/cross-validation-70289113a072.
[9] J. Hale. (-03-04T14:00:51.391Z). *Scale, Standardize, or Normalize with Scikit-Learn*. Available: https://towardsdatascience.com/scale-standardize-or-normalize-with-scikit-learn-6ccc7d176a02.
[10] M. Hart, "Massively Parallel Hyperparameter Optimization on AWS Lambda," 2019. Available: https://medium.com/@hichaelmart/massively-parallel-hyperparameter-optimization-on-aws-lambda-a7a24b1970c8?fbclid=IwAR0oFHHZX1Jd4MRxnCgNDqTCsqFfXZaq2aot5ZrgGLazg5BdwTlpnYzRt-8.

Table of Contributions

	Research	**Presentation**	**Implementation**	**Report**
Abhinav Behal abeh957	25%	33.33%	60%	20%
Matthew Frost mfro529	50%	33.33%	30%	10%
Molly Farrant mfar672	25%	33.33%	10%	70%