Hyper-Stacked: Scalable and Distributed AutoML Approach for Big Data

Ryan Dave
School of Computer Science
University of Nottingham
Nottingham, UK
enyrd2@nottingham.ac.uk

Juan S. Angarita-Zapata
DeustoTech, Faculty of Engineering
University of Deusto
Bilbao, Spain
js.angarita@deusto.es

Isaac Triguero
Computer Science and Artificial Intelligence
University of Granada
Granada, Spain
triguero@decsai.ugr.es

***Abstract*—The emergence of Machine Learning (ML) has altered how researchers and business professionals value data. Applicable to almost every industry, considerable amounts of time are wasted creating bespoke applications and repetitively hand-tuning models to reach optimal performance. For some, the outcome may be desired; however, the complexity and lack of knowledge in the field of ML become a hindrance. This, in turn, has seen an increasing demand for the automation of the complete ML workflow (from data preprocessing to model selection), known as Automated Machine Learning (AutoML). Although AutoML solutions have been developed, Big Data is now seen as an impediment for large organisations with massive data outputs. Current methods cannot extract value from large volumes of data due to tight coupling with centralised ML libraries, leading to limited scaling potential. New approaches are therefore required. This paper introduces Hyper-Stacked, a novel AutoML component built natively on Apache Spark. Hyper-Stacked combines multi-fidelity hyperparameter optimisation with the Super Learner stacking technique to produce a strong and diverse ensemble. Integration with Spark allows for a parallelised and distributed approach, capable of handling the volume and complexity associated with Big Data. Scalability is demonstrated through an in-depth analysis of speedup, sizeup and scaleup.**

***Index Terms*—AutoML, Big Data, Apache Spark, Supervised learning, Binary classification.**

I. Introduction

Automated Machine Learning (AutoML) is an emerging area in Machine Learning (ML) that seeks to automate the ML workflow from data preprocessing to model validation [1]. Such automation provides robust AutoML methods that enable people, with either little or no specialised knowledge, to integrate ML solutions into the daily activities of business organisations.

With AutoML, ML solutions are now easily accessible by expert and non-expert ML users. Those methods usually search for the most suitable ML methods and their best hyperparameters (known as the Combined Algorithm Selection and Hyperparameter problem, CASH problem [2]) using an online search strategy; that is, a process takes place after the input dataset has been provided. This online search can be purely based on optimisation approaches that test different promising combinations of algorithms from a predefined base of ML classifiers to minimise or maximise a performance measure [3]. Alternatively, there are AutoML methods whose online search is complemented with learning strategies like meta-learning [4]. These techniques first extract meta-features of the input dataset at hand (e.g., number of instances, features, classes). From these meta-features, meta-learning identifies good candidates of pipeline structures from a predefined knowledge base that stores meta-features for different datasets and ML models that are likely to perform well on them. Then, the candidate models are typically used for a warm-start an optimisation approach. In addition, other AutoML methods use ensemble learning to build diverse sets of classifiers from predefined portfolios of ML algorithms [5], [6]. These ensemble approaches have proven to be more robust than other AutoML methods, such as the case of Auto-Gluon, which is the state-of-the-art in AutoML thanks to its ensemble learning strategy based on multi-layer stacking [7].

In recent years, both the number of data sources and the scale of such data have seen the global data volume increase exponentially [8]. This data is referred to by the term 'Big Data' [9]. The challenge of computing large amounts of data resides in the simple principle that volume increases complexity [10]. Furthermore, as the training time of an ML algorithm is heavily dependent on the number of data points, efficient ML algorithms must exploit parallelism to achieve sufficient scalability. Without this, operations on large datasets become infeasible.

Open source AutoML solutions fail to handle the size and variety of Big Data [11]. Popular tools are often coupled with ML libraries that rely on centralised data and processing and will only work on a single machine [12]. Consequently, these cannot scale up as a single machine is limited in terms of parallelism due to restrictions in hardware. Some commercial products claim to scale AutoML workloads over multiple nodes; however, many fail to take advantage of superior Big Data frameworks, such as Apache Spark. Those that are built to run on Spark implement outdated solutions, e.g. TransmogrifAI (grid search) [13], or attempt to integrate it into such frameworks as a second thought (e.g., H2O's Sparkling Water that is an interface between Spark and H2O), which does not fully leverage the framework's abilities. Significant progress has been made in areas of AutoML; however, Big Data solutions are comparatively underdeveloped. Therefore, a gap exists for a novel solution that implements an efficient,

scalable solution that can run natively on Spark. In that context, we take the AutoML state-of-the-art and build further upon the stacking ensemble concept by integrating k-fold cross validation to conceive a Super Learner [14], which is implemented natively on Apache Spark. Thus, we can propose an AutoML method to handle Big Data based on the simple concept that more diversity is introduced with more models, leading to increases in stacking performance.

This article introduces a novel approach to scalable AutoML in Big Data, named Hyper-Stacked. It combines the strength of the Super Learner stacking ensemble, the efficiency of Greedy K-Fold hyperparameter optimisation, and Apache Spark's scalability. The main contributions of this work are:

- A novel AutoML component design, Hyper-Stacked, is presented. This design efficiently integrates greedy k-fold and the Super Learner stacking approach to produce a high-performant ensemble. The approach automates the search for a diverse set of models and combines them to bolster the overall performance. Results showed that the ensemble consistently outperforms the best-performing individual model.
- This approach was implemented natively on Apache Spark to produce a distributed and scalable model capable of dealing with the volume, variety and complexity associated with Big Data. To validate these claims, parallelism and scalability were critically evaluated in speedup, sizeup and scaleup experiments. Results showed that Hyper-Stacked can handle data growth significantly better than sequential processing and single node parallelisms.

The rest of this paper is structured as follows. Section II presents background and related work about AutoML with special emphasis on the CASH problem, Ensemble learning, Meta-learning, and Spark. Then, Section III introduces Hyper-Stacked. Section IV exposes the experimental framework, and Section V presents and analyses the results obtained. Finally, conclusions are discussed in Section VI.

II. Background and Related work

A. Problem definition

In AutoML, when the task of algorithm selection is combined with hyperparameter optimisation, it is often referred to as the CASH problem. It can be defined as the follows [5].

Let γ denote the loss that an algorithm $A^{(j)}$ returns on $D^{(i)}_{valid}$ when trained on $D^{(i)}_{train}$, with hyperparameters λ. Given the set of algorithms A, their respective hyperparameters Λ, and sets of cross validation folds D_{train} and D_{test}, CASH focuses on determining the joint algorithm $A^{(j)}$ and hyperparameter $\Lambda^{(j)}$ that minimises the loss γ.

$$A^*, \lambda = \underset{A^{(j)} \in A, \lambda^{(j)} \in \Lambda}{argmin} \frac{1}{K} \sum_{i=1}^{k} \gamma \left(A^{(j)}_{\lambda}, D^{(i)}_{train}, D^{(i)}_{test} \right) \quad (1)$$

As an alternative to using the $argmin$ operator with respect to a single algorithm $A^{(j)}$, we can instead construct a set E, where E represents an ensemble. In this instance, more than a single algorithm can be chosen and individual predictions are combined to produce a final output. The new representation is presented below.

$$A^*, \lambda = \underset{E \subseteq A, H \subseteq \Lambda}{argmin} \frac{1}{K} \sum_{i=1}^{k} \gamma \left(E_H, D^{(i)}_{train}, D^{(i)}_{test} \right) \quad (2)$$

B. CASH methods

This section introduces the most representative approaches to solve the CASH problem presented in equation 1.

Black box optimisation approaches

The most basic optimisation technique for hyperparameter tuning is grid search. Grid search involves performing an exhaustive search given a subset of the hyperparameter space [15]. For example, an algorithm may require the tuning of 3 distinct parameters. Parameter values are selected in uniform or exponential intervals to form sets of candidate parameters. The algorithm then iterates over each possible combination of the three parameter subsets to return the best. In this context, a clear limitation exists, as it remains essentially a brute force approach. As the number of distinct parameters increase, the number of possible combinations will increase exponentially and therefore is not viable for larger datasets.

An improvement on grid search emerged, known as random search. Random search aims to trial a number of random hyperparameter configurations and has been proven to return models that are equivalent or better, within a fraction of the computation time [16]. The method of randomly sampling the space, rather than brute force allows the exploration of a lager search space given the same computational budget.

Alternatively, Bayesian Optimization (BO) provides an adaptive approach to black box optimisation. It works by first building a surrogate model (a cheaper approximation function). Then, the uncertainty in that surrogate is then evaluated. Finally, desirable sample spaces are proposed by an acquisition function defined from this surrogate. As samples are selected, the surrogate is updated iteratively and the uncertainty is re-quantified. Auto-Sklearn, a python based AutoML library, adopts this as the primary optimisation method. However, in the context of Big Data, the standard approach to BO fails to succeed in high dimensional environments and new approaches are required [17]. AutoML tools, such as Auto-SKlearn, are yet to incorporate these new approaches.

Multi-Fidelity approaches

Multi-fidelity optimisation seeks to speed up the optimisation process by using performance estimates from lower-fidelity models [18]. In general, these techniques rely on first training lower-fidelity models (e.g., models trained with a low computational budget) to reveal promising configurations. These models can then be allocated additional computational budget to continue training and give a "higher" fidelity model. It is relevant to mention that in this approach the computational budget must be easily measurable (e.g., training time).

State-of-the-art multi-fidelity methods are built on the successive halving algorithm that was first proposed by Karnin et al. [19]. Successive halving initially randomly samples a set of hyperparameter configurations and models are trained with a specified budget and evaluated to return a metric. The configurations are ranked based on the metric and the worst performing half is discarded. Then, rounds of successful halving are performed until one configuration remains. In this sense, successive halving can give the effect of early stopping which can heavily reduce computation.

Other relevant methods under the multi-fidelity approach are Hyperband and Greedy k-fold. Hyperband [20] works on the main principle of multi-fidelity by randomly distributing budget values and performing rounds of successive halving. This allows the exploration of different convergence behaviours and ensures that configurations are not discarded too early. On the other hand, Greedy k-fold [21] applies a similar mechanism to k-fold cross validation. It works by first evaluating a single fold for all configurations, then proceeds with a greedy approach. Again, low fidelity models are initially trained, but rather than performing rounds of evaluations, it pursues only the most promising candidate model. Evaluations in the original paper show results to perform significantly better than the successive halving approach, on average 70 % faster.

The main limitation of multi-fidelity evaluation is that using low fidelity approximations to perform early stopping may remove an optimal configuration. This is not usually seen as a concern for most, as the performance speedup often heavily outweighs the approximation error [1]. Efficiency is important to combat volume when limited to a computational budget.

C. Ensemble learning

This section provides an overview of the existing literature surrounding ensemble learning to solve Equation 2.

Majority Voting and Stacking

Majority voting remains one of the simplest methods in ensemble learning. Within this ensemble approach, a set of base (heterogenous or homogenous) classifiers are all trained on the same data set. When making a prediction, every data point results in a prediction from every classifier. Then, a final prediction is made by selecting the class that had the most "votes" from the set of classifiers.

Stacking learning builds upon the weighted ensemble by training a meta-learner on model predictions. Specifically, a set of base learners are first trained on a training set and each output a prediction. Afterward, predictions are aggregated to construct a new dataset where each data point holds the predictions from each base model. Thus, the meta-learner can be trained to learn complex behaviours of the base models. Rather than a user-given weighting, the meta-learner will determine the importance of base models empirically.

A significant limitation exists with the stacking method. It is easily susceptible to overfitting. If a single base model is seen to overfit, the meta-learner may result in a heavy reliance on that same model and will harm the overall model performance. This limitation has been reduced by techniques that centre around the use out of fold predictions, i.e. base models will predict on unseen data. These methods commonly produce the highest accuracy out of any individual model or ensemble methods and hence remain popular [22], but are often overlooked due to their added complexity. Stacking methods are especially applicable here as research has shown that stacking can handle high-dimensional datasets [23], a common attribute of Big Data.

Super learner

Super Learner builds further upon the stacking ensemble by integrating k-fold cross validation, such as is done in H2O's AutoML framework [6]. In the H2O implementation, heterogenous ensembles (different types of learners) are used. Then, a mix of random and fixed grids are used to diversify, and two super learners are trained. One of the super learner is optimised for model performance by including all model configurations. The latter is based on the simple concept that more diversity is introduced with more models, leading to an increase in stacking performance.

Meanwhile, the second super learner is optimised for production uses. In this case, this super learning considers only the best model from each algorithm to output faster predictions [6]. The benefit of this twofold approach is that the two super Learners perform "asymptotically as well as best possible weighted combination" [24], and therefore both will perform at least as effectively as the best performing base model.

K-fold repeated bagging

In the context of AutoML, there is another competitive approach coupled with ensemble learning, which is K-fold repeated bagging. This approach can be observed in the AutoML framework named Auto-Gluon [7]. Auto-gluon implements an improved method to prevent overfitting in their approach to multi-layer stacking. Multi-layer stacking passes predictions through multiple sets of models, rather than a single set of base models. These ensembles have the potential to perform better than single layer models however tend to suffer more from overfitting as the effect is amplified through layers.

To combat this, k-fold repeated bagging was introduced. K-fold repeated bagging makes additional o-of-fold predictions on n different random partitions of the training data and takes an average. The value of n is determined by dividing the total allotted time between an estimate of the time taken for a given partition. The overall approach is therefore heavily dependent on the given budget. It is important to say that Auto-gluon was shown to outperform H2O's framework and 99 % of data scientists in a Kaggle benchmark, however it remains a centralised approach. In a distributed context, hurdles such as the non-trivial task of time estimation need to be considered.

D. Meta-learning

In this section, we will be focusing on aspects of meta-learning that rely on learning from prior tasks rather than outputs coming within the same task. The interest reader can

be referred to [4] to consult all relevant approaches of meta-learning within AutoML. To learn from prior tasks, a learning algorithm may be run a number of times and the related data from the training (model evaluations, hyperparameter configurations, training time etc.) can be stored as features in a new dataset [1]. It raises the layer of abstraction above traditional ML in two main approaches.

The first approach is learning from model evaluations. Advances in this area have demonstrated effective results in warm-start optimisation [25]. Warm-start optimisation removes the exploration of search spaces that have been explored in similar tasks and provides a starting point for hyperparameter optimisation. Conversely, the second approach is learning from task properties, which looks at the CASH problem from a different perspective. Instead of tuning every algorithm, the search space can be reduced by selecting a few of the most promising. Learning tasks can be characterized by meta-features and a meta-model can provide a way of associating these meta-features to a subset of algorithms based on prior experience. This has produced promising results in the context of Big Data, when combined with multi-fidelity optimisation [12]. The primary limitation is that only a finite amount of information can be captured in the meta features [4].

E. Spark

Apache Spark is an open-source, distributed processing framework used for large scale workloads. This section provides relevant concepts related to Apache Spark.

Data Locality

Typically, to deal with larger datasets, the user may be required to scale their resources, that is, additional memory or cores may be added. Nevertheless, this approach of scaling up on a single node, known as vertical scaling, fails to continue to scale as it is limited to the hardware capacity and eventually will reach a hard upper limit.

An alternative paradigm to vertical scaling is horizontal scaling, which allows nodes to be added to an existing pool of resources. As more machines can be introduced, the user is no longer bound by the hardware limits. The traditional approach to high performance computing (HPC) relies on communication between storage nodes and compute nodes. In this sense, a bottleneck exists between storage and compute in data intensive jobs, as network I/O becomes the limiting factor and node computation remains low and unused [26].

The bottleneck mentioned above can be resolved by distributing data across the compute nodes and storing it on local disks. This allows each node to perform operations on their subset of the data, reducing cross-switch network traffic and leading to a performance gain [26]. This is the concept of data locality and is essential to the scalability of data processing.

In summary, regardless of how the code is written, scalability is also heavily dependent on the architecture and where your data is situated. In this context, Spark implements data locality to facilitate the efficient compute of operations.

Parallelism

When performing parallel operations on shared data, the data itself must also support parallelism. Concurrent, or parallel data structures allow data to be accessed by multiple threads. Spark implements resilient distributed datasets (RDDs) to accomplish data parallelism by organising data as a collection of partitions that can be held over one or more machines [27]. This can then be operated in parallel via a low-level API, through actions and transformations.

For example, data may be partitioned into 20 distinct partitions, across 2 nodes in a cluster. Spark can then run a single task per partition in that RDD concurrently, up to the number of cores in that cluster. If each machine has 4 cores, it is possible to run 8 concurrent tasks on 8 partitions. This allows scalability as tasks can be run independently across hundreds of nodes in a cluster.

Spark ML

Spark ML is Apache Spark's ML library that implements ML algorithms and utility functions. This scalable library is in some sense a basic AutoML library. It allows a pipeline to string together pre-processing operations and a Cross-Validator class to perform grid search and return the best model. There currently exists no available ensemble learning methods, aside from common ML algorithms such as Random Forest and Gradient Boosted Trees that are ensembles as themselves.

Spark ML is important to scalability as implementations overcome the curse of modularity. The curse of modularity states that there is an assumption behind ML algorithms that the data can fit, in its entirety, in memory on a single machine [28]. In other words, some algorithms have been developed using modular strategies that when used outside of the scope of in-memory data, will break. This explains why many popular libraries are inherently unable to scale. Opposite to such situation, Spark ML implements these algorithms in a way that can be broken down and distributed across multiple machines.

III. Hyper-Stacked: A Scalable and Distributed Approach to AutoML for Big Data

In this section, we introduce Hyper-Stacked [1]. First, Section III-A motivates the need for the proposed method. Then, Section III-B presents the general architecture of Hyper-Stacked and details of its inner workflow.

A. Motivation

As we stated before, the core aims of existing AutoML methods are (1) High Computational Efficiency, (2) Good Performance, and (3) Reduced Human Interaction. Nevertheless, the current solutions suffer from the following issues, which have motivated the design of Hyper-Stacked.

- **Centralised data approach**: Open source AutoML solutions fail to handle the size and variety of Big Data [11]. Popular tools are often coupled with ML libraries that rely on centralised data and processing and will

[1] https://github.com/jsebanaz90/Hyper-Stacked

only work on a single machine [12]. Consequently, these are unable to scale as a single machine is limited in terms of parallelism due to restrictions in hardware. Some commercial products claim to scale AutoML workloads over multiple nodes; however, many fail to take advantage of superior Big Data frameworks, such as Apache Spark.

- **Optimisation and reduced scalability**: When optimisation deals with small- or medium-size datasets, many algorithms can be generated, tuned and tested because the complexity of the learning task at hand is influenced by its data size. On the other hand, the latter may stop happening as the data size grows. In this scenario, it is harder to tune and test multiple algorithms, as the optimisation becomes expensive and the set of candidate ML methods could decrease, affecting the final solutions' performance.

Considering the motivations presented above, we want to conceive a new AutoML method that relies on a distributed and scalable approach. This will enable us to deal with the volume, variety and complexity associated with Big Data. In this sense, we introduce Hyper-Stacked, a new AutoML method based on greedy k-fold and Super Learner stacking to produce a high-performant ensemble. The approach automates the search of a diverse set of models and combines them to bolster the overall performance, which allows to path the way towards three main goals of AutoML: (1) High Computational Efficiency, (2) Good Performance, and (3) Reduced Human Interaction. Firstly, the Super Learner was chosen as the base mechanism as it hosts the high performance of stacking (goal 2) and overcomes the common problem of overfitting. Secondly, The overarching concept of Hyper-Stacked focuses on finding strong heterogeneous learners amongst the search space to achieve a high stacking performance, while keeping the number of base models low to remain efficient (goal 1). Thirdly, Hyper-Stacked aims to succeed in both through effective hyperparameter optimisation. In doing so, we are guaranteed to automatically (goal 3) return the best individual model (the aim of the CASH problem) and an ensemble that returns an equal or higher predictive performance.

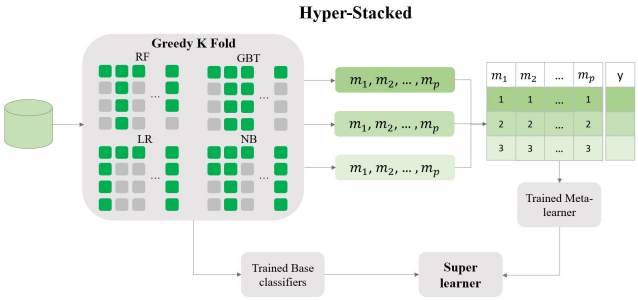
In summary, Hyper-Stacked combines the strength of the Super Learner stacking ensemble, the efficiency of Greedy K-Fold hyperparameter optimisation, and the scalability of Apache Spark. To the best of our knowledge, at this time, the Super Learner has not yet been implemented on Apache Spark and no solution yet exists that combines the Super Learner and Greedy k-fold optimisation.

B. Hyper-Stacked's design and workflow

Hyper-Stacked was implemented as a Scala package that can run on top of a distributed spark cluster. We also used MLlib, a Spark's ML library, to make practical machine learning scalable and manageable. From this library, we selected the following classifiers to be part of the algorithms that can be part of the ensemble built in the inner structure of Hyper-Stacked: Random Forest (RF), Gradient Boosted Trees (GBT), LinearSVC (LSVC), Logistic Regression (LR), and Naïve Bayes (NB). Besides, the meta-learner is chosen from this portfolio of methods.

The architecture of Hyper-Stacked is shown in Figure 1. Besides, pseudocode 1 presents the step-by-step followed by Hyper-Stacked's Super learner and Greedy k-fold components; wherein lines 1-19 illustrate the generation of the base models and the meta-learner selection, lines 20-25 represent the training process of a Hyper-Stacked model, and lines 26-28 summarize how the final predictions are made. More details of this process are presented as follows.

Fig. 1. General architecture of Hyper-Stacked based on based on greedy k-fold and Super Learner stacking



We provide Hyper-Stacked with a list of descriptor tuples instead of providing a set of candidate models. Each tuple contains a learning algorithm and the number of random hyperparameter configurations to generate. Random parameters are generated and fed into the greedy k-fold method for each algorithm type. The latter allows us to run greedy k-fold multiple times, returning different model types. The specified number is important to vary the number of configurations to account for the different hyperparameter search spaces an algorithm may have.

The result of the original greedy k-fold would contain the hyperparameter configuration of the best model, and a model could then be trained on the full dataset with those optimal hyperparameters. In Hyper-Stacked, we return models that were trained during hyperparameter optimisation, reuse them and the k-fold data splits later. Instead of a configuration, a list of trained models is returned, one for each fold. The final result of greedy k-fold will be a list of lists of fold models, where each element represents a different hyperparameter configuration that was fully evaluated; this is flattened for ease of iteration. In addition, to minimise the movement of data, each fold is considered one at a time, and we iterate through each hyperparameter configuration before continuing to the next fold. Each fold model will output a prediction, and we aggregate them in the same way as the original to train the meta learner.

To output a prediction from the Super Learner, features are first passed into each base model to construct a set of features for the meta-model. The meta-model can then produce a final output based on the results of the base models. The meta-classifier can therefore learn complex behaviours of the base classifiers. In this case, rather than a user-given weighting,

as defined in the weighted ensemble, the meta-classifier will empirically determine the importance of base classifiers.

Finally, pseudocode 1 also shows the automatic selection of the meta-learner. Hyper-Stacked removes the reliance on the user to choose a meta-learner by performing an additional round of cross-validation on the list of available learning algorithms. This was added to comply with the original core goals of AutoML outlined earlier in the paper and minimise human assistance and interaction. It is essential to perform this selection the same way it was for the original training set because we do not know a priori which algorithm will perform best as the meta-learner.

Algorithm 1: Super Learner with Greedy k-Fold

Data:
$D_{train} \longleftarrow$ training dataset
$D_{test} \longleftarrow$ test dataset
n $\longleftarrow$ number of k folds
T $\longleftarrow$ list of tuples *(a,r)* where *a* is a learning algorithm and *r* is the number of random parameters to generate for that algorithm
M $\longleftarrow$ a list of learning algorithms included in meta-learner selection
Result: Set of predictions corresponding to input D_{test}

k_folds $\longleftarrow$ divide D_{train} into *n* number of approximately equal partitions;
$L_{best} = []$;
for *t in T* **do**
  $C_a \longleftarrow$ generate random candidate models *(a,r)*;
  $\lambda_{best} \longleftarrow$ **do** Greedy K-Fold(C_a) where $|\lambda_{best}| \geq 1$;
  append λ_{best} to L_{best};
end
flatten L_{best};
for k_i *in k_folds* **do**
  $k_{valid} \longleftarrow k_i$;
  $k_{train} \longleftarrow$ remaining *k* folds;
  for *l in* L_{best} **do**
    OOF $\longleftarrow$ predict on k_{valid} with *l* and store result and label;
  end
  $i_{OOF} \longleftarrow$ concatenate *OOF* (D_{train} *. length x l . length*);
end
meta-features $\longleftarrow$ union all i_{OOF};
meta_k_folds $\longleftarrow$ divide *meta-features* into *n* number of approximately equal partitions;
$m_{best} \longleftarrow$ **do** K-fold Cross-Validation(*meta_k_folds*);
base_models = [];
for *l in L* **do**
  base_model $\longleftarrow$ train *l* on D_{train};
  append *base_model* to *base_models*;
end
meta_model $\longleftarrow$ train m_{best} on *meta-features*;
base_layer_output $\longleftarrow$ **for** *base_model* in *base_models* **do** transform D_{test} with *base_model*;
predictions $\longleftarrow$ transform *base_layer_output* with *meta_model*;
return *predictions on* D_{test}

IV. Experimental Design

This paper seeks to answer whether it is possible to design a distributed and scalable AutoML to deal with Big Data in supervised learning tasks. To accomplish such a purpose, we introduce the Hyper-Stacked method. In particular, this method is tested in binary supervised classification problems and determines its performance in three crucial Big Data metrics: Speedup, Scaleup, and Sizeup.

This section shows the factors and issues related to the experimental study. First, we provide details of the problems chosen for the experimentation (Section IV-A). Then, we introduce details about the big data architectures considered to test Hyper-Stacked in Section IV-B. Finally, we present the three experiments on Speedup, Scaleup, and Sizeup metrics.

A. Binary supervised learning problems

For this experimentation we chose four representative datasets, which are shown in Table I. These datasets represent binary classification problems and their composition vary on size and dimensions.

It is important to mention that the number of instances in these datasets may be lower than traditionally seen in current real-world Big Data. The intention is to keep the number of instances in a permissible range to run on a single node and return in a 'reasonable' time. In reality, a single node would take a significant amount of compute time and could take days, weeks or months. With mid-large size datasets, we can run trials comfortably on different-sized clusters to demonstrate the scalability and strength of the approach. These datasets are deemed suitable for ML g problems by the research community and are commonly used in benchmarks and research papers.

B. Experimental setups

All experiments done with Hyper-Stacked were set up on the Databricks platform, which allows clusters of variable size and configurations to be easily instantiated. To measure the time of experiments, we implemented a logger object. Specifically, the time before is recorded, then the function is executed, and the time is once again recorded to calculate the difference between them finally. The log function is only used around larger functions as small operations are likely to return inaccurate results due to lazy evaluation. A run-time limit of ten hours was applied to all trials, and the leader board of model performances was recorded for every trial.

The specifications for the three chosen cluster sizes to be used in these experiments are summarised in Table II. All cluster sizes contain a single driver node with 4 cores with 14GB memory. The memory of each worker remains the same (28GB).

TABLE I
BINARY CLASSIFICATION DATASETS

Dataset	Number of Training instances	Number of features
FLIGHT	516513	29
SUSY	3500066	18
HEPMASS	4899792	28
HIGGS	7701355	28

TABLE II
SPECIFICATIONS FOR THE SPARK CLUSTERS TO BE USED IN THE EXPERIMENTS

Cluster	Number of workers	Number of cores per worker	Total number of cores
1	1	1	1
2	1	8	8
3	3	8	24

C. Experiment Speedup, Sizeup, Scaleup

In this set of experiments, the primary aim is to demonstrate the scalability, efficiency and effective parallelism of Hyper-Stacked. Therefore, we step up experiments around speedup, sizeup and scaleup metrics to accomplish such an aim. These experiments focused on measuring the relative change as an element of the system changes. The three experiments are described as follows.

- In speedup, the size of the data remains constant and the number of cores are increased. Speedup shows how much faster the same data can be processed with *n* cores instead of 1. This metric can be estimated by calculating the ratio of the time taken for a sequential execution versus a parallel execution. For the experiments carried out in this work, the data is kept at 33 % as this value was achievable by all cluster sizes.
- In sizeup, the number of cores remains constant and the size of the data is increased. Sizeup allows us to see how time scales with increasing intervals of data size. This metric can be found by calculating the ratio of execution time between an initial dataset versus a dataset n-times larger. For this study, the intervals were chosen to be 10 %, 20 %, 40 % and 80 %.
- In scaleup, both the number of cores and size of the data are increased (by the same factor). Scaleup combines the two to measure how a program performs as a system gets larger and has to process larger datasets. The following configurations were chosen: 100/24 ($\approx$ 4.16 %) with 1 core, 100/3 ($\approx$ 33.33 %) with 8 cores and 100 % with 24 cores.

V. ANALYSIS OF RESULTS

This section analyses the experimentation results from the following angles.

- Speedup: to evaluate the ability of Hyper-Stacked of improving its execution time through parallelism (using additional cores).
- Sizeup: to assess the ability of Hyper-Stacked to handle increasing amounts of data within a parallel enviroment.
- Scaleup: to evaluate the ability of Hyper-Stacked to handle both increase the amount of data and the size of the system. It can be found by calculating the ratio of execution time between an initial dataset and system, versus a dataset m-times larger with an m-times larger system.

A. Speedup

Figures 2a and 2b show the speedup for each dataset with a single core, eight cores, and twenty four cores. As it can be seen, the speedup achieved for the FLIGHT dataset was only 1.899. This means that despite parallelising over 24 cores, the component was not able to achieve even double the speed. Furthermore, there was almost no speedup between eight cores and twenty-four (speedup increase of 0.084).

In contrast, the speedup achieved for the HIGGS dataset was 4.413, but again, despite the addition of sixteen cores, the speedup increase was 1.125. This can be clearly seen in Figure 2a where comparisons were done againts linear speedup.

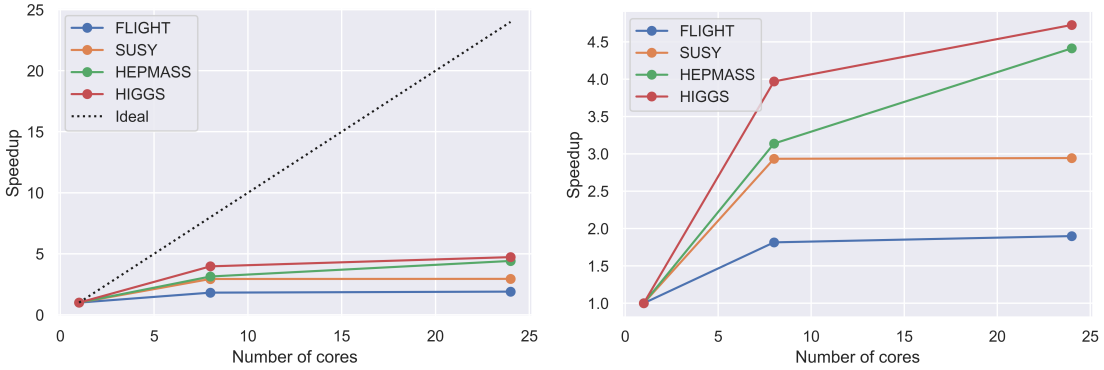
Through these results, it is hard to pinpoint the main cause of the lack of speedup. It should be noted that between the datapoints at eight and twenty-four cores is the introduction of a distributed cluster (increase from 1 to 3 worker nodes). Despite this, the speedup is not expected to be as low as observed in a Spark application. Through additional investigation, the training of base models (layer one models) and meta models (layer two models) were compared in terms of speedup. Figure 3 shows that the speedup of these are almost identical, failing to identify any existing limiting. component.

Furthermore, Figure 2b shows a clearer comparison between the datasets. A trend can be clearly seen as the larger datasets (HIGGS and HEPMASS) achieve a significantly better speedup than the smaller datasets (FLIGHT and SUSY). In fact, they are sorted in size order. As a result of these findings, an additional run was performed, which is shown in Figure 4.

In this experiment, the size of the data was increased from 33 % to 100 % of the dataset. As seen in Figures 4a and 4b, a greater difference is seen when using a larger amount of data. The speedup analysis in Figure 4a shows a speedup increase from 4.413 (seen previously) to 10.210. This can be considered a reasonable speedup and significantly better than seen when using 33 % of the data. Therefore, we can say the speedup can be heavily dependent on the size of the data, and clearly, the amount of the data used in the initial experiments was insufficient. It is possible that some rate-limiting factors exist within the component; however without additional experiments using larger datasets, it is difficult to determine at this time.

B. Sizeup

The results in Figures 5 and 6 depict the sizeup for each dataset. Specifically, Figure 5a shows the sizeup results for the smallest of the four datasets (FLIGHT dataset). In the results can be observed that Cluster 1 is able to execute eight times the size of the original data with a size up of 3.753. In comparison, Cluster 3 is able to execute eight times the size of the original data with a size up of 1.964. The latter means that although

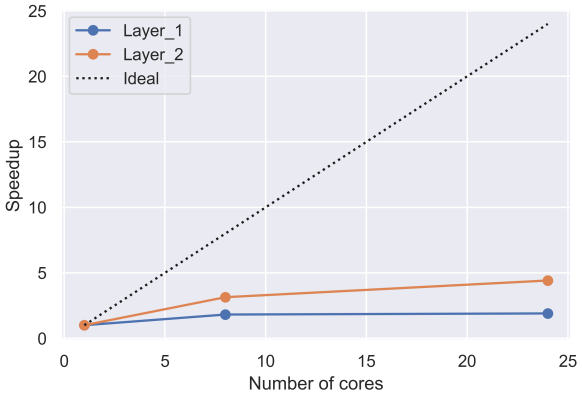


(a) Hyper-Stacked vs Linear speedup

(b) Hyper-Stacked's speedup by dataset

Fig. 2. Speedup analysis in supervised binary classification problems

Fig. 3. Speedup comparison of Layer 1 and Layer 2 models for HIGGS dataset



the data increased eightfold, the execution time of Cluster 1 increased by around a factor of 4, whereas Cluster 3 only increased by a factor of 2.

In the other datasets, the overall data size increases and Cluster 1 is unable to execute within the given budget. Using the SUSY dataset 5b, Cluster 1 is still able to execute 40 % of the data with a sublinear size up of 3.753. Unfortunately, problems arise when executing HEPMASS and HIGGS (Figures 6a and 6b). Due to the 10-hour constraint, we are only able to gather two datapoints, 10 % and 20 %. For the 20 % size up, cluster 1 begins to exceed a linear size up for HEPMASS and HIGGS achieving a size up 2.617 and 2.310 respectively. The gradients of the lines shown in Figures 6a and 6b, clearly show that as the datasets become larger, systems with sequential processing are unable to deal with the growth of data.

Additionally, both clusters 2 and 3 are able to return the results for all datasets in a reasonable time. Size up values between clusters 2 and 3 are similar with smaller datasets, as seen in Figures 5a and 5b. These two clusters are also consistently similar at low data percentages with all of the datasets. However, the difference is seen when the growth of the larger datasets reaches 80 %. In the largest dataset (HIGGS), shown in Figure 6b, an eightfold data size increase caused the execution time of Cluster 2 to increase by a factor of 3.602, whereas Cluster 3 only increased by a factor of 2.024. This shows that the parallelism within Hyper-Stacked allows greater sized clusters to effectively handle the growth of data.

C. Scaleup

Figure 7 shows the scaleup analysis for Hyper-Stacked. It displays a comparison to the ideal scaleup value. The ideal value is where the execution time is kept equal as the system and data size grows by the same factor, and in reality is unattainable. Although Hyper-Stacked does not achieve an 'ideal' result, all results appear to taper off in an acceptable range between 0.684 and 0.792. The decrease between 8 and 24 cores is assumed to be partially down to the fact the processing is now distributed and communication (shuffle read and write) between workers is necessary.

From the graphs presented, we can see that the scaleup exhibits a similar trend to speedup. The lines appear sorted in size order, HIGGS achieving a significantly better efficiency value than FLIGHT. FLIGHT achieves a scale up of 0.774 then 0.684, whereas HIGGS achieves a better result of 0.995 then 0.792. Similar to speedup, it is hard to determine the true metric through these experiments as the sizes of the datasets chosen seem to be insufficient. Nevertheless, the findings show that Hyper-Stacked is still shown to be scalable when increasing the size of the data, number of cores and number of nodes.

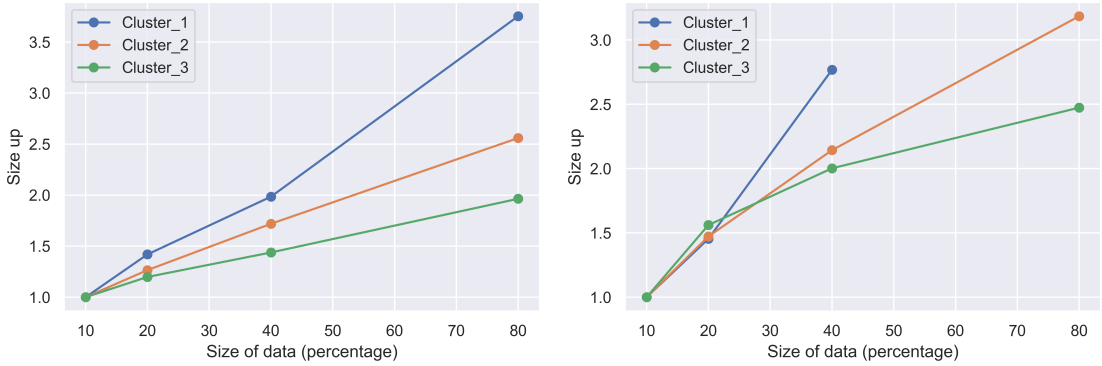
VI. Conclusions

In this work, we introduced Hyper-Stacked. This new AutoML method combines the strength of the Super Learner stacking ensemble, the efficiency of Greedy K-Fold hyper-parameter optimisation, and the scalability of Apache Spark. Hyper-Stacked was implemented natively on Apache Spark to produce a distributed and scalable model capable of dealing with the volume, variety and complexity of Big Data. Parallelism and scalability were critically evaluated in speedup, sizeup and scaleup through different experiments to validate

(a) Training time comparison between clusters in HIGGS dataset

(b) Speedup comparison in HIGGS dataset

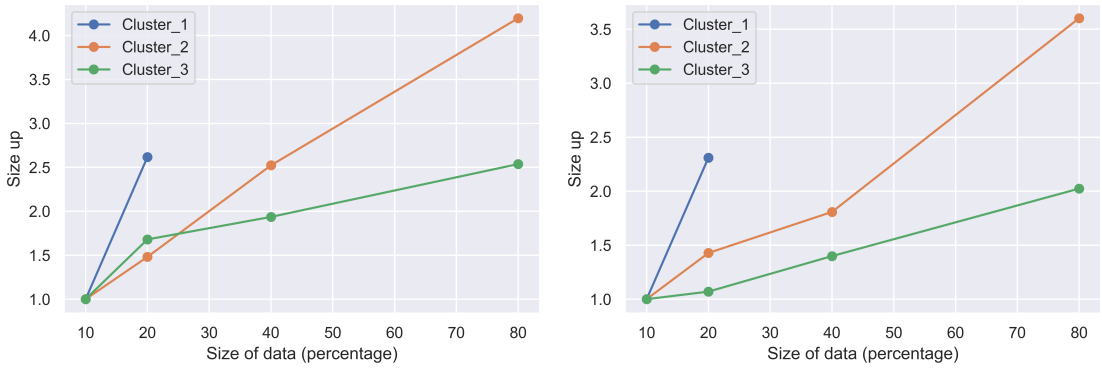
Fig. 4. Speedup experiment with HIGGS dataset using different data sizes



(a) Sizeup comparison for the FLIGHT dataset

(b) Sizeup comparison for the SUSY dataset

Fig. 5. Sizeup comparison for FLIGHT and SUSY datasets



(a) Sizeup comparison for the HEPMASS dataset

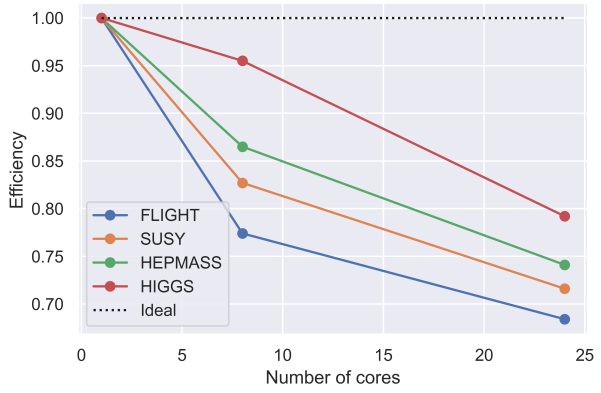
(b) Sizeup comparison for the HIGGS dataset

Fig. 6. Sizeup comparison for HEPMASS and HIGGS datasets

the general architecture of Hyper-Stacked. The experiments focused on binary classification problems, using datasets varying in size and dimensions.

From the results obtained, we extracted interesting conclusions. A limitation was uncovered during the speedup experiments in Section V-A, where the addition of workers resulted in a minimal speedup. This can be explained by Amdahl's law, which states that the performance increase from parallelisation cannot exceed the inverse of the non-parallelisable element of work [29]. The mechanism that Spark uses to distribute work (setting up a job on the driver, scheduling and data shuffles) results in overhead, i.e. a nonzero amount of non-parallelisable work. The relative amount of parallelisable work is then bounded by the size of the dataset, as larger datasets

Fig. 7. Hyper-Stacked vs ideal scaleup by dataset



reduce the significance of the overhead. As Hyper-Stacked was developed to perform efficiently with Big Data, it may be preferable to use an alternative tool for smaller datasets that fit in the memory of a single machine.

In future work, we will follow different paths. First, the implementation itself, despite only being currently applicable to binary classification, can be easily extended to regression and multiclass classification using different Spark ML algorithms. An evaluation of all three problem types would provide additional insight into the applicability of this AutoML approach and its ability to generalise to different problems. Second, larger, more complex datasets could be approached to check the robustness and scalability of Hyper-Stacked in multiclass supervised learning problems.

ACKNOWLEDGEMENT

This work is supported by projects A-TIC-434-UGR20 and PID2020-119478GB-I00).

REFERENCES

[1] F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018.

[2] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-WEKA," in *Proceedings of the 19th International conference on Knowledge discovery and data mining*, aug 2013, pp. 847–855.

[3] R. S. Olson, N. Bartley, R. J. Urbanowicz, and J. H. Moore, "Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 2016, pp. 485–492.

[4] J. Vanschoren, "Meta-learning," F. Hutter, L. Kotthoff, and J. Vanschoren, Eds. Springer, 2018, pp. 39–68.

[5] M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter, "Efficient and robust automated machine learning," in *Advances in Neural Information Processing Systems*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 2962–2970.

[6] E. LeDell and S. Poirier, "H2o automl: Scalable automatic machine learning," *7th ICML Workshop on Automated Machine Learning (AutoML)*, 2020.

[7] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, and A. Smola, "Autogluon-tabular: Robust and accurate automl for structured data," *arXiv preprint arXiv:2003.06505*, 2020.

[8] Q. Yao, M. Wang, Y. Chen, W. Dai, H. Yi-Qi, L. Yu-Feng, T. Wei-Wei, Y. Qiang, and Y. Yang, "Taking Human out of Learning Applications: A Survey on Automated Machine Learning," *CoRR*, 2019.

[9] B. Christiansen, "Ensemble averaging and the curse of dimensionality," *Journal of Climate*, vol. 31, no. 4, pp. 1587–1596, 2018.

[10] C. Parker, "Unexpected challenges in large scale machine learning," in *Proceedings of the 1st International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, ser. BigMine '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1–6.

[11] J. Waring, C. Lindvall, and R. Umeton, "Automated machine learning: Review of the state-of-the-art and opportunities for healthcare," *Artificial Intelligence in Medicine*, vol. 104, p. 101822, 2020.

[12] A. Abd Elrahman, M. El Helw, R. Elshawi, and S. Sakr, "D-smartml: A distributed automated machine learning framework," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, 2020, pp. 1215–1218.

[13] K. Moore, , K. F. Kan, L. McGuire, M. Tovbin, M. Ovsiankin, M. Loh, M. Weil, S. Nabar, V. Gordon, and V. Patryshev, "Transmogrifai," https://github.com/salesforce/TransmogrifAI, 2017.

[14] J. van der Laan Mark, P. E. C, and H. A. E., "Super learner," *Statistical Applications in Genetics and Molecular Biology*, vol. 6, no. 1, pp. 1–23, 2007.

[15] P. B. Liashchynskyi and P. B. Liashchynskyi, "Grid search, random search, genetic algorithm: A big comparison for nas," *ArXiv*, vol. abs/1912.06059, 2019.

[16] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. 10, pp. 281–305, 2012. [Online]. Available: http://jmlr.org/papers/v13/bergstra12a.html

[17] R. Moriconi, M. P. Deisenroth, and K. S. Sesh Kumar, "High-dimensional bayesian optimization using low-dimensional feature spaces," *Machine Learning*, vol. 109, no. 9–10, p. 1925–1943, 2020.

[18] A. March and K. Willcox, "Constrained multifidelity optimization using model calibration," *Structural and Multidisciplinary Optimization*, vol. 46, 07 2012.

[19] Z. Karnin, T. Koren, and O. Somekh, "Almost optimal exploration in multi-armed bandits," in *Proceedings of the 30th International Conference on International Conference on Machine Learning*. JMLR.org, 2013, p. III–1238–III–1246.

[20] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *Journal of Machine Learning Research*, vol. 18, pp. 1–52, 2018.

[21] D. S. Soper, "Greed is good: Rapid hyperparameter optimization and model selection using greedy k-fold cross validation," *Electronics*, vol. 10, no. 16, 2021.

[22] B. Pavlyshenko, "Using stacking approaches for machine learning models," in *2018 IEEE Second International Conference on Data Stream Mining and Processing (DSMP)*, 2018, pp. 255–258.

[23] S. R. Sharma, B. Singh, and M. Kaur, "A novel approach of ensemble methods using the stacked generalization for high-dimensional datasets," *IETE Journal of Research*, vol. 0, no. 0, pp. 1–16, 2022.

[24] E. C. Polley and M. J. van der Laan, "Super learner in prediction," in *Causal Inference for Observational and Experimental Data, Chapter 3. New York, Springer*, 2010.

[25] K. Swersky, J. Snoek, and R. P. Adams, "Multi-task bayesian optimization," in *Advances in Neural Information Processing Systems*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., vol. 26. Curran Associates, Inc., 2013.

[26] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, 2012, pp. 419–426.

[27] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. USA: USENIX Association, 2010, p. 10.

[28] K. A. Kumar, J. Gluck, A. Deshpande, and J. Lin, "Hone: "scaling down" hadoop on shared-memory systems," *Proceedings of the VLDB Endowment*, vol. 6, no. 12, p. 1354–1357, aug 2013.

[29] S. Pei, M.-S. Kim, and J.-L. Gaudiot, "Extending amdahl's law for heterogeneous multicore processor with consideration of the overhead of data preparation," *IEEE Embedded Systems Letters*, vol. 8, no. 1, pp. 26–29, 2016.