Hyperon: An Online Hyperparameter Tuning Approach for Data Stream Learning

Sadia Tabassum and Leandro L. Minku
School of Computer Science, *University of Birmingham*
Birmingham, United Kingdom
sxt901@alumni.bham.ac.uk, L.L.Minku@bham.ac.uk

***Abstract*—Predictive models built using machine learning algorithms usually involve a number of hyperparameters that can significantly affect their performance. While many approaches for hyperparameter tuning have been investigated for offline learning, there is little work in the context of online data stream learning. Hyperparameter tuning for online data stream learning can be particularly challenging, due to possible changes in the underlying distribution of the problem. Such changes can result in the best hyperparameter choice varying over time, requiring efficient, real-time online adaptation. However, existing online hyperparameter tuning approaches are limited to specific models, are susceptible to local optima, rely on fixed hyperparameter grids, or on concept drift detection methods. We propose a novel online hyperparameter tuning approch for data stream learning called Hyperon to overcome these issues. Hyperon intertwines online data stream learning with a steady-state evolutionary algorithm, enabling efficient and effective hyperparameter optimisation over time. Experiments on 10 real world data streams show that Hyperon is able to significantly improve the predictive performance of the underlying online data stream learning approach in a computationally efficient manner.**

***Index Terms*—Hyperparameter tuning, online data stream learning, evolutionary computation.**

I. Introduction

Predictive models built using machine learning algorithms usually involve a number of hyperparameters. Unlike ‘model parameters’ which are learned during the training of the model, ‘hyperparameters’ need to be carefully set before running the algorithm. Hyperparameters are critical for predictive models as different hyperparameter choices can lead to different predictive performances [2]. Therefore, several hyperparameter tuning algorithms have been proposed.

Hyperparameter optimisation techniques, such as Grid search and Random search [3], are predominantly designed for offline settings. These methods operate under the assumption that the entire dataset is stationary, fully accessible, and does not change over time. However, many machine learning problems are data stream learning problems, where data arrive sequentially over time, requiring online learning algorithms capable of learning new examples without requiring retraining on all previous data. Some examples are software defect prediction [4], fraud detection [5], electricity price prediction [6]. Some applications such as “TinyML” with on-device learning also require online learning with strict memory constraints [7].

Online data stream learning presents unique challenges for hyperparameter tuning. As data distributions in online environments often evolve over time (concept drift), the optimal hyperparameter configurations may change over time, and static settings chosen at the beginning can become suboptimal, leading to significant performance degradation. Although not all data stream learning problems exhibit such variability [8], the need for adaptive hyperparameter tuning methods is crucial to maintain optimal predictive performance in several real-world applications [9], [10].

Existing studies on hyperparameter tuning for data streams [9], [11]–[15] are limited to specific models or tasks, rely on periodic batch retraining, rely on concept drift detection methods, or impose high computational overhead, restricting their practical applicability. Additionally, in online data stream learning, the conventional use of static, pre-defined training and validation sets becomes infeasible due to the continuous arrival of data, limited memory resources, the presence of concept drift, and the need for real-time model adaptation. Therefore, an efficient online hyperparameter tuning approach is required which is adaptive and able to operate under these constraints. *This paper proposes a novel online hyperparameter tuning approach for this purpose.* It answers the following research questions:

- RQ1: How to automatically tune hyperparameters in an online and adaptive manner? To what extent can this help improving predictive performance?
- RQ2: How high is the computational cost of online hyperparameter tuning approach? In particular, how feasible is the new approach in terms of runtime?

To answer the above RQs, we propose a novel online evolutionary hyperparameter tuning approach for online learning, that explores the hyperparameter space and updates the predictive model with appropriate hyperparameters over time. Our experiments based on 10 real-world data streams show that the proposed online hyperparameter tuning approach significantly improved the predictive performance of online models compared to an upper bound of performance achievable by online Grid Search (RQ1). Although our proposed approach has a higher computational cost compared to Grid search, the computational cost was still minimal, making it suitable for

This work was partly funded by a School Scholarship provided in support of the EPSRC grant EP/R006660/2. It was preliminary included in the phd thesis [1].

adaption in practice (RQ2).

The overall contributions of the paper are following:

- Proposal of a novel online hyperparameter tuning approach based on evolutionary optimisation.
- Empirical demonstration that online tuning improves predictive performance over time.
- Evidence that the proposed approach is computationally feasible for practical deployment.

This paper is further organised as follows. Section II presents the problem formulation. Section III presents related work. Section IV introduces our online hyperparameter tuning approach. Section V presents the investigated datasets. Section VI explains the experimental setup for answering the RQs. Section VII explains the results of the experiments. Section VIII presents the conclusions and future work.

II. Problem Formulation

We consider online data stream learning problems where $S_{train} = \{(\mathbf{x}_t, y_t)\}_{t=0}^{\infty}$ is a sequence of training examples received over time, $\mathbf{x}_t$ are the input features, y_t is its corresponding label, and t is the time step when the label of the training example arrived. The labels arrive with a delay compared to the input features, a phenomenon known as verification latency [16]. In particular, a given example $(\mathbf{x}_t, y_t)$ is used for training and hyperparameter tuning $\delta_t > 0$ days after it is used for testing. Each example $(\mathbf{x}_t, y_t)$ comes from an underlying joint probability distribution that was active when the input features were produced. This distribution may change over time, a phenomenon known as concept drift [17]. This may cause the best hyperparameter choice to potentially change over time. Our problem is to optimise the hyperparameter choice for online learning models at each time t so that the models will perform well for the current underlying probability distribution. The online hyperparmeter tuning algorithm may rely on a limited-size window of recent training examples including $(\mathbf{x}_t, y_t)$.

III. Related Work

A. Offline Hyperparameter Tuning

Several offline hyperparameter tuning approaches exist. They search for optimal hyperparameter values based on a pre-defined validation set. Grid Search is a common brute force method for this purpose [4]. It is easy to implement, but inefficient when the number of hyperparameters increases, being unsuitable for high dimensional spaces [18]. Random search was proposed to be used for hyperparameter tuning by Bergstra et al. [19] and was shown to be able to find models that are as good or better within less computational time than grid search, being more efficient in high dimensional spaces.

Gradient descent methods have also been used for hyperparameter tuning. A study [20] used reversed stochastic gradient descent with momentum to compute the gradient with respect to all continuous hyperparameters. Another study [21] used approximate gradient instead of true gradient because computing an exact gradient can be computationally demanding. However, these techniques can get stuck in local optima.

Other hyperparameter tuning approaches are based on metaheuristic optimisation, which are well known to improve robustness to local optima. Genetic algorithm is widely used for hyperparameter optimisation [22]. HyperOpt [23] adopts Bayesian optimisation for both model selection and hyperparameter optimisation. It uses a surrogate model to approximate the objective function and an acquisition function to direct the samples towards areas where an improvement over the current best observation is likely [24], [25].

Reinforcement learning has also been used for hyperparameter tuning. Bandit-based algorithms such as Successive Halving [26] and Hyperband [27] have shown success. Successive halving eliminates half of the poorly performing hyperparameters and allows more budget to the better-performing half to be evaluated in next iteration. However, allocating budget efficiently is a concern for Successive Halving. Hyperband deals with this by achieving a good trade-off between the number of hyperparameters and their allocated budget.

All studies above require a pre-existing validation set to evaluate different hyperparameter choices. When dealing with data streams, such choices may become inadequate over time.

B. Batch-Based Hyperparameter Tuning

Batch-based hyperparameter tuning approaches are similar to offline ones, but enable hyperparameter values to be changed over time. These approaches process data in batches / chunks and perform tuning by applying an offline tuning approaches on a validation set that is either sampled from each batch, or consists of the previous batch of training examples.

Jie et al. [14] proposed a hyperparameter optimisation technique called Hypertube. They introduced a set of initial settings based on a micromini-batch training mechanism and used a Genetic Algorithm to create new candidate hyperparameters. As it re-optimises from scratch for each new batch, it has high computational cost and wastes potentially useful knowledge about past hyperparameter values that may have performed well. Joy et al. [15] proposed a Bayesian optimisation for hyperparameter tuning in data streams [15], by dividing the data stream into batches and generating hyperparameter combinations for a given batch based on their performance on the previous batches.

Despite these approaches being applicable to data streams, they still apply offline tuning procedures for each batch. They also have to wait for a new batch to arrive before hyperparameter values can be updated, delaying adaptation.

C. Online Hyperparameter Tuning

Hyperparameter tuning in online learning scenarios can be challenging due to the constraints that it presents such as continuous learning from data streams and dealing with concept drifts caused by changes in the data generating process.

In data stream learning, there are two types of approaches can be identified based on concept drift handling: non-explicit approaches, which adapt hyperparameters without relying on explicit drift detection, and explicit approaches, which rely on a drift detection mechanism to trigger adaptation.

There are several existing non-explicit hyperparameter tuning approaches in data stream learning. A study [11] proposed an incremental learning-model selection method for support vector machine (SVM) for data streams, which involves incremental tuning of the hyperparameters of the SVM. However, it is limited to SVMs. Another study [28] proposed an algorithm that computes the hyper-gradient on the fly when a new datum is observed and then the average of the historical hyper-gradient is computed. However, it is mainly tailored for Kernel Ridge Regression, and might not generalise for other models or complex data where the hyper-gradients is unreliable.

Lin et al. [12] performed online hyperparameter optimisation for auto-augmentation inspired by a forward hyper-gradient approach. However, their approach still needs a separate fixed, pre-defined validation set, making it less suitable for fully online learning. In addition, it only searches within a fixed set of step changes (augmentation), limiting its ability to find new augmentation strategies dynamically. An online supervised hyperparameter tuning procedure based on online grid search called Dycom was proposed in [13]. This approach operates in a strict online manner, but is limited to a grid of hyperparameter configurations. Interestingly, there are no online random search tuning methods in the literature.

There are also some explicit approaches. Veloso et al. [9] proposed an approach called SSPT, by adjusting the Nelder-Mead optimisation procedure to work with data streams. However, the Nelder-Mead algorithm can get stuck in local optima or non-stationary points [29], and can be very inefficient for high dimensional problems [30], [31]. An approach based on differential evolution called MESSPT was proposed in [32] to overcome this problem. However, it generates and maintains very few new configurations over time, potentially requiring a large number of iterations to find good values, and thus being more suitable for very high speed data streams. However, these approaches rely on a concept drift detection method to trigger a hyperparameter optimisation phase, being prone to false positive and negative detections.

Overall, though addressing some challenges in online hyperparameter tuning, existing online methods suffer from model-specific designs, limited flexibility or fixed hyperparameter grids, susceptibility to local optima, high computational costs, or reliance on very high speed data, or on drift detection methods. These gaps highlight the necessity for an adaptive, model-agnostic, and resource-efficient online tuning approach that can dynamically adjust hyperparameters in real time.

IV. Proposed Approach

This section proposes a novel online hyperparameter tuning approach – Hyperon, which is a non-explicit hyperparameter tuning approach, i.e., it does not rely on drift detection. The proposed algorithm itself is presented in Section IV-A, while Section IV-B explains the evolutionary algorithm Asynchronous Differential Evolution with Adaptive Correlation Matrix (ADE-ACM) [33] adopted within Hyperon and how Hyperon interacts with it.

A. *Hyperon*

Hyperon is an online meta-heuristic algorithm that searches for good hyperparameter configurations over time through the data stream. It maintains a pool of online learning models trained based on different hyperparameter combinations. The hyperparameters used by the models in this pool are evolved based on a steady-state evolutionary algorithm. Hyperon considers each candidate solution $\vec{x}_i$ [1] in the evolutionary algorithm as a vector containing the hyperparameters of the underlying machine learning algorithm being tuned. Hyperon's hyperparameter optimisation iterations are intertwined with the learning of incoming examples, such that the evolutionary algorithm is run in an online manner together with the training of the online learning models. The models in the pool are evaluated based on a sliding window containing the most recent labelled examples. Such evaluation is used both as the fitness function for the evolutionary algorithm and to determine the most promising hyperparameter configuration, which is then used for predicting new incoming examples.

The evolutionary algorithm adopted by Hyperon is ADE-ACM [33]. We have adopted this algorithm due to it being quasi-parameter-free from the user's perspective, which makes it a suitable option to adopt in Hyperon to search for new hyperparameter combinations. Many evolutionary algorithms are generation-based strategies where evolutionary operations are performed on a large portion of the population to generate offspring solutions. Typically the number of generated offspring is the same as the population size. Offspring solutions then replace old solutions of the population. This makes such algorithms impractical for continuous optimisation. We used steady-state ADE-ACM, where evolutionary operations are performed on a (few) selected member(s) of the population instead of a large portion of the population. Typically, only one new solution can be inserted into the population, allowing the new better solution to take part in the evolution process without any time lag [33]. Vavak et al. [34] showed that the steady version outperforms the generational genetic algorithm regarding the convergence speed for real-time/non-stationary environments. Hence, a steady-state optimisation algorithm would be more suitable to tune hyperparameters in online data stream learning.

Algorithm 1 shows the pseudocode of the Hyperon approach. It generates an initial set of hyperparameter combinations H (line 1). This initial set is randomly sampled from a grid of possible values. One of the combinations in H is enforced to be the default combination typically used with the underlying machine learning algorithm. This avoids poor initial performance resulting from the use of completely random hyperparameter values in the beginning of the data stream. A pool M of n models is created, where each model corresponds to a hyperparameter combination in H (line 2).

When a training example $d^t = (\mathbf{x}_t, y_t)$ from the training data stream S_{train} arrives, each model m_p in the pool M makes a prediction, which is stored in a list $\hat{Y}$ (line 4).

[1]Note that this is different from the input feature vector $\mathbf{x}_i$.

Algorithm 1 Proposed Hyperon Approach

Input: S_{train} = training data stream b = test project index, w = waiting period, h = default hyper-parameter combination, $M : \{m1, m2, ..., m_n\}$ = Pool of models,
Algorithm Hyperparameters: $minAge$ = minimum age of the model to be old enough, $windowSize$ = max size of the most recent instance window, $eaInterval$ = interval to run ADE-ACM, n = max number of models
Variables: $InstWindow$ = window of most recent instances, $ErrorWindow$ = window containing the prediction errors of classifiers on $InstWindow$, $fitlist$ = hyperparameters of models in M with their fitness.

```
1: Generate set of random hyper-parameter combination and add default hyper-parameter combinations $H : \{h1, h2, ..., h_n\} \cup \{h\}$
2: Initialise model pool $M$ consisting of $n$ models $\{m_1, m_2,$ ... ..., $m_n\}$ using $n$ hyperparameter combinations from $H$

3: for each training example $d_t = (\mathbf{x}_t, y_t)$ in $S_{train}$ do
4:     $\hat{Y}$=List of predictions on $\mathbf{x}_t$ by each model $m$ in $M$
5:     for each model $m_p$ in $M$ do
6:         Update $ErrorWindow[m_p]$ with $(\hat{Y}[m_p] != y_t)$
7:         $fitlist[m_p]$=$fitnessCalc(m_p)$
8:         Train model $m_p$ with $d_t$
9:     end for
10:    if $t$ % $eaInterval == 0$ then
11:        $fit_{h'}$ = ADE-ACM($fitlist$), $m_{new}$ and $errors_{m_{new}}$ are the new model and its mistakes
12:        if $fit_{h'}$ is better than the old enough worst model $m_{wrst}$ in the pool $M$ then
13:            Del $m_{wrst}$ from $M$, $ErrorWindow$, $fitlist$
14:            Add $m_{new}$, $fit_{h'}$ and $errors_{m_{new}}$ to $M$, $fitlist$ and $ErrorWindow$, respectively
15:        end if
16:    end if
17:    Yield the best fitness model from $M$ as the one to be used for testing purposes
18: end for

19: function MODELEVAL($m_{new}$)
20:     $offspringError$ = []
21:     for each training example $d_j = (\mathbf{x}_j, y_j)$ in $InstWindow$ do
22:         $\hat{y}_j$ = $m_{new}$.predict($\mathbf{x}_j$)
23:         $offspringError[m_pnew]$ = $(\hat{y}_j != y_j)$
24:         $m_{new}$.train($d_j$)
25:     end for
26:     Return $fitnessCalc(m_{new})$
27: end function

28: function FITNESSCALC($m_p$)
29:     $fitlist[m_p] = 0$
30:     for each $err$ in $ErrorWindow[m_p]$ do
31:         $fitlist[m_p]$ += $err$
32:     end for
33:     $fitlist[m_p]$ /= $windowSize$
34: end function
```

Each model m_p in M is also associated to an error window $ErrorWindow$ containing the classifier's mistakes (1 for mistake and 0 otherwise) on the most recent $windowSize$ instances of S_{train}, and a list $fitlist$ to keep track of the hyperparameter values and fitness of that model. Fitness is the rate of mistakes in the error window for a model (line 28). Next, $ErrorWindow$ and $fitlist$ are updated for each model in M (line 6-7). Then, each model m_p in M is trained with d_t (line 8). It is important that the training on d_t is done *after* this example is used for estimating the error for *ErrorWindow*, to avoid underestimations of the error.

After that, Hyperon checks whether it has seen a certain number of training examples, which is denoted by $eaInterval$ (line 10). At every $eaInterval$, ADE-ACM is run to create a new hyperparameter combination (line 11). Hyperon's function $ModelEval$ enables ADE-ACM to evaluate the fitness of this new combination by creating a new predictive model by prequentially evaluating-then-training it on each training example stored in the $InstWindow$. If ADE-ACM produces a hyperparameter configuration with better fitness than its parent configuration, it returns the new hyperparameter combination and its fitness value $fit_{h'}$. Its corresponding model and errors on $InstWindow$ are named m_pnew and errors $errors_{m_pnew}$, respectively. Then, Hyperon selects the model with the worst fitness m_{wrst} from M, and compares it with $fit_{h'}$ (line 12). If $fit_{h'}$ is better than the fitness of m_{wrst}, then m_{wrst} and its corresponding positions in $ErrorWindow$ and $fitlist$ are deleted (line 13). Next, the new model m_pnew and its corresponding $fit_{h'}$ and errors $errors_{m_pnew}$ are added to M, $fitlist$ and $ErrorWindow$ (line 14).

Hyperon always yields the best model from M whenever a new training example is received (line 17). Therefore, even when $eaInterval > 1$, the approach can still continuously choose the best among the models in the population.

B. ADE-ACM

This section explains ADE-ACM [33] and how Hyperon interacts with it to optimise hyperparameters in an online manner. The pseudocode of ADE-ACM is shown in Algorithm 2. ADE-ACM maintains a population P of vectors $\vec{x}_i$. In the case of Hyperon, P is taken as $fitlist$ from Hyperon. A "target" vector $\vec{x}_i$ is selected randomly from P. Mutation is applied on it to create a mutant vector (line 1) as follows:

$$\vec{v}_i = \vec{x}_i + F * (\vec{x}_{best} - \vec{x}_i) + F * (\vec{x}_{r1} - \vec{x}_{r2}), \qquad (1)$$

where F is the mutation factor, $\vec{x}_i$ is the target vector, $\vec{x}_{best}$ is the current best vector in the population, $\vec{x}_{r1}$ and $\vec{x}_{r2}$ are two randomly selected vectors different from $\vec{x}_i$ and $\vec{x}_{best}$. F is sampled from a Cauchy distribution as below using a location parameter $\mu \cdot F$ and a scale parameter $\sigma \cdot F$:

$$F = randc(\mu F, \sigma F). \qquad (2)$$

This mutation operator moves the vector $\vec{x}_i$ in the direction of the best vector to encourage improvement in fitness and in another random direction to encourage diversity, resulting in exploration of novel areas of the search space.

Algorithm 2 ADE-ACM pseudocode [33], [35]

Input: current population P, dimension D, learning rate clr=0.01, mutation factor F=0.5, location parameter uF=0.5, scale parameter σF=0.1, approximated correlation matrix C

```
1: Randomly select a target vector $\vec{x}_i$ from $P$
2: Create a mutant vector using Eq. 1 with $\mu F$, $\sigma F$
3: $S$ = sample correlation matrix from $P$
4: Select a random coordinate $m$ and random threshold $c_{thr}$
5: Get set of variables correlated with $m$ from $C$ using Eq. 3
6: Create a trial vector using Eq 4
7: if (trial_vector fitness < target_vector fitness) then
8:     $C = (1 - clr) * C + clr * S$
9:     Update $\mu F = (1 - clr)\mu F + clr * L2(F)$ // L2(F)
       is a contraharmonic mean of a set of all scale factors
       associated with the current population
10:    return trial_vector and its fitness
11: else
12:    return null
13: end if
```

Instead of using a crossover rate, ADE-ACM modifies the crossover operator to take into account pairwise dependencies between the variables. The current population P is used to calculate a sample correlation matrix (line 3). The approximation C of the correlation matrix is cumulatively adapted through successful evolutionary steps (lines 7–8). The advantage of covariance matrix adaptation is that it can tackle ill-conditioned and non-separable problems. By analysing all elements of the adaptive correlation matrix C, ADE-ACM identifies correlated variables, which define a subspace in the search domain.

For a random index m and a randomly selected threshold c_{thr}, the algorithm identifies variables correlated with m as (line 4-5):

$$\{I_m\} = \forall j :\mid c_{mj} \mid > c_{thr}. \tag{3}$$

Crossover moves all components of mutant vector $\vec{v}_i$ whose indices are in $\{I_m\}$ to a trial vector $\vec{u}_i$, and other components are taken from the target vector $\vec{x}_i$ (line 6) as shown below, eliminating the need for a crossover rate:

$$u_{i,j} = \begin{cases} v_{i,j} & \text{if } j \epsilon \{I_m\} \\ x_{i,j} & \text{otherwise.} \end{cases} \tag{4}$$

Next, we compare the fitness of the trial vector and the target vector. We get the fitness of the target vector from $fitlist$ from Hyperon, that contains fitness values of solutions in the current population. The fitness of the trial vector is computed over $InstWIndow$ using the $ModelEval$ function in line 19 of the Hyperon algorithm, where m_{new} is a new model initialised with the hyperparameters defined in the trial vector $\vec{u}_i$. If the trial vectors' fitness is better than the fitness of the target vector, C and μF are updated, and the trial vector is returned as the new hyperparameter combination (line 7-13).

Zhang et al. [36] investigated different variants of differential evolution and found that automatic adaptation of the control parameters as done above can be effective and helpful in improving the algorithm's robustness.

V. Datasets

For this study, we used 13 datasets extracted from open-source software repositories [16]. Table I summarises key dataset characteristics. They represent sequential software change data described by 13 numeric and 1 binary feature, with associated labels indicating whether they are defect-inducing changes, allowing us to evaluate online hyperparameter tuning approaches in a real-world, evolving data stream scenario. These datasets have been shown to present temporal dependencies and evolving data distributions [4], [16], [37], making them ideal for studying online hyperparameter tuning. Additionally, they are affected by class imbalance, as defect-inducing changes are less frequent, and verification latency, since labels for defects are only available after later corrective commits. These properties make the datasets highly representative of dynamic, non-stationary data streams. Three of these datasets were used for the sole purpose of tuning the hyperparameters of the hyperparameter tuning approaches themselves, enabling us to choose default values to be adopted for all other 10 data streams, which were used in our analyses.

VI. Experimental Setup

A. Compared Approaches

To answer RQ1, the predictive performance achieved by Hyperon is compared against the following approaches:

- Offline Grid Search (Fixed Configuration): This baseline uses a conventional grid search performed on the initial 3,000 data instances. The hyperparameter combination that achieves the best average predictive performance during this initial phase is then fixed throughout the data stream. This approach is widely used in data stream learning due to its simplicity and reflects standard practice in many existing studies [4], [16], [37], [38]. It is included to check whether automatically tuning hyperparameters over time through Hyperon would outperform standard practice.
- Theoretical Upper Bound (Online Grid Search with Perfect Hindsight): This is a hypothetical and non-realizable scenario, where at each time step, the best-performing hyperparameter configuration is selected with perfect hindsight from the predefined grid of 243 combinations. While not applicable in practice, this setting serves as a theoretical upper limit, representing the best possible performance achievable by methods relying on fixed configuration sets. It also represents the theoretical maximum achievable performance of Dycom [13], the most recent non-explicit online hyperparameter approach that operates in a strict online manner without requiring a predefined validation set.
- A dummy classifier that predicts class 1 or 0 uniformly at random was also used to support the analysis. This is important as it has been found that some past studies were obtaining results worse than random guess [39].

TABLE I: An overview of the datasets (adapted from [4])

Dataset	Total	# Examples	% Class 1 examples	Class 1 Median Verification Latency δ_t (days)	Time Period
Tomcat	18907	5207	27.54	200.5798	27-03-2006 - 06-12-2017
JGroups	18325	3153	17.21	116.1565	09-09-2003 - 05-12-2017
Spring-integration	8750	2333	26.66	415.1201	14-11-2007 - 16-01-2018
Camel	30575	6255	20.46	28.1947	19-03-2007 - 07-12-2017
Brackets	17364	4047	23.31	14.454	07-12-2011 - 07-12-2017
Nova	48989	12430	25.37	88.5615	28-05-2010 - 28-01-2018
Fabric8	13106	2589	19.75	39.1833	13-04-2011 - 06-12-2017
Neutron	19522	4607	23.6	82.5097	01-01-2011 - 27-12-2017
Npm	7920	1407	17.77	111.514	29-09-2009 - 28-11-2017
BroadleafCommerce	15010	2531	16.86	42.5818	19-12-2008 - 21-12-2017
Datasets used for tuning Hyperon's Hyper-Parameters					
Django	27364	11614	42.44	150.3181	13-07-2005 - 27-09-2019
Rails	74651	15085	20.21	153.2868	24-11-2004 - 27-09-2019
Vscode	56399	1187	2.1	37.48	13-11-2015 - 26-10-2019

Note: the class 0 verification latency is always 90 days.

The online data stream learning approach to be tuned in the experiments was All-in-One Oversampling Rate Boosting (AIO-ORB), as it has shown to perform well for the datasets used in our experiments [37]. It has 5 hyperparameters expected to significantly affect performance (ma_window_size, th, $l0$, $l1$ and m). To answer RQ2, computational cost of Hyperon is compared with Offline Grid search, which will provide deeper understanding on the feasibility of the proposed approach. All experiments were based on thirty runs.

B. Predictive Performance

We adopt the Geometric Mean (G-Mean) of Recall on Class 0 ($Recall_0$) and Recall on Class 1 ($Recall_1$) as measures of predictive performance. These metrics were computed prequentially and using a fading factor to enable tracking changes in predictive performance over time, as recommended for problems that may suffer concept drift [40]. The fading factor was set to 0.99 as in [16]. It is worth noting that $Recall_0$=1-$FalseAlarmRate$, i.e., false alarms are taken into account through $Recall_0$ in the G-Mean. These metrics were chosen because they have been recommended as unbiased metrics for online class imbalance learning [17], [41], [42].

C. ORB, Grid Search and Hyperon Hyperparameters

Hyperon itself has 4 hyperparameters, which are tuned by executing 5 runs of the algorithm with a grid search, using 3 datasets (Django, Rails and Vscode), which are different from the 10 datasets used for evaluation as mentioend in Section V. Being a hyperparameter tuning approach, Hyperon is only a valid approach if it can itself work well using default hyperparameter values. Otherwise, the problem of tuning model hyperparameters would merely shift to tuning Hyperon's own hyperparameters. Therefore, these additional datasets were used exclusively to determine default values for Hyperon, ensuring that the algorithm can be applied effectively in other data stream scenarios without requiring further manual configuration. The grid search was based on total of 144 different combinations from the following set of values $minAge = \{50, 500, 1000, 3000\}$, $windowSize = \{50, 500, 1000, 3000\}$, $eaInterval = \{1000, 3000, 5000\}$ and $ns = \{20, 50, 70\}$ for each of the 3 datasets using 7000 initial training examples. We assign a rank for each combination for each dataset based on average G-Mean of 5 runs. The combination with best average rank across the 3 datasets was selected: $\{500; 50; 3000; 70\}$. Once this hyperparameter combination was set, it remained fixed over time, i.e., Hyperon is used to tune the underlying machine learning algorithm, and not to tune itself.

For initialisation of the pool in Hyperon, n hyperparameter combinations were sampled from the following set of AIO-ORB hyperparameter values, where the grid was based on [16] and the values in bold are the default values adopted in [4], [16], [37], [38]: $ma\_window\_size = \{50, \mathbf{100}, 200\}$, $th = \{0.3, \mathbf{0.4}, 0.5\}$, $l0 = \{5, \mathbf{10}, 15\}$, $l1 = \{6, \mathbf{12}, 18\}$ and $m = \{\mathbf{1.5}, 3, 5\}$. These combinations were also used for the compared Grid Searches.

D. Statistical Tests and Effect Size

The predictive performances obtained using Hyperon will be compared with Grid Search across data sets using the Scott-Knott procedure, which ranks the models and separates them into groups with different rankings, where smaller rankings are better rankings. Non-parametric bootstrap sampling is used to make the test non-parametric, as recommended by Menzies et al. [43]. As explained by Demsar [44], non-parametric tests are adequate for comparison across datasets. In addition, the Scott-Knott test adopted in this paper uses the Vargha-Delaney A12 effect size to rule out statistically significant but small differences in performance. Specifically, Scott-Knott only performed statistical tests to check whether groups should be separated if the A12 effect size was not insignificant [43]. If the A12 effect size was insignificant, groups were *not* separated. We will refer to Scott-Knott based on Bootstrap sampling and A12 as Scott-Knott.BA12.

We also report the A12 effect sizes against the Hyperon approach for each dataset individually to support the analysis, indicating how large the differences in performance are.

Wilcoxon rank sum test was also used to support comparisons between Hyperon and the Upper Bound with a confidence level of 95% (i.e., p-value $<$ 0.05). This test was chosen for being a non-parametric test to compare two groups.

VII. Experiment Results

A. *RQ1: Online Hyperparameter Tuning's Improvements in Predictive Performance*

To evaluate the predictive performance of Hyperon, we compared it with the Offline Grid Search and the dummy classifier. Table II shows the average G-Mean of 30 runs. According to Scott-Knott.BA12, Hyperon ranked the best (lowest) overall ranking across datasets. A12 effect sizes of Grid Search and dummy classifier against Hyperon were always large. Fig. 1 further shows the G-Mean over time for Hyperon in black and for Offline Grid Search in red. The plots show that Hyperon's superior performance was consistently higher than that of Offline Grid Search. Notably, even when Offline Grid Search found strong hyperparameter choices for the initial portion of the stream, these configurations often became worse over time. This shows that offline approaches in general would be unsuitable, even if they were not restricted to a grid of hyperparameter values like the Offline Grid Search, as they cannot change the hyperparameter choices over time.

However, it is possible that if an online grid search like Dycom [13] was adopted, it would be able to select better combinations from the grid over time, leading to higher predictive performance. For an impartial comparison, we consider the Online Grid Search Upper Bound. The Wilcoxon rank sum test shows that the difference between G-Means of Hyperon and Upper Bound is significant (p-value=9.605e-10 < 0.05), with Hyperon outperforming it. Hyperon achieved up to 6.9% (for Tomcat) absolute improvement in G-Mean over the Upper Bound. Fig. 1 shows the G-Mean over time for all of the grid's hyperparameter combinations in green. At each point in time, the Upper Bound corresponds to the maximum G-Mean among these. The plots show that Hyperon's superior performance (in black) was consistently higher than that of the Upper Bound. The comparison against Upper Bound suggests that Hyperon's ability to investigate new hyperparameter combinations not restricted to a grid is helpful. In particular, Hyperon employs an evolutionary algorithm to generate new hyperparameter combinations over time in an online fashion. This enables it to dynamically explore and exploit the hyperparameter search space as data continuously arrives, rather than being restricted to a fixed set of configurations.

To obtain a more in depth understanding of this aspect, Fig. 2 shows new hyperparameter combinations generated by Hyperon over time that were selected for use in predictions at least at some point in time, due to their top average G-mean. New combinations whose models obtained better performance than those from previous combinations are shown as different coloured points. Even though there is a chance that the differences in performance between these configurations are due to the new models (with the new configurations) being trained on the most recent window of data while old models (with the old configurations) were also trained with additional examples received before this window, previous work has shown that models trained only on the most recent window of data did not perform well on these data sets compared to ORB trained on all examples received so far [16]. This suggests that the better performance achieved by the models corresponding to the new coloured combinations is likely due to the new configurations themselves being better for the current time period, rather than due to the different training data.

From Fig. 2, we can see that, for 5 datasets, Hyperon managed to generate at least one new useful combination, and for the other 5 datasets, it was able to create more than one (up to 7) new hyperparameter combinations (for Nova Fig. 2f) over time that performed better than the existing ones. As Hyperon creates a new combination at every $eaInterval = 3000$ time steps, for larger data sets such as Camel (30575 examples) and Nova (48989 examples), the number of new generated combinations is larger. Hence, the possibility of generating better new combinations is higher compared to smaller data. However, even for some of the smaller datasets like JGroups (18325 examples), Hyperon managed to create 5 new better hyperparameter combinations. These results suggest that Hyperon was able to find new useful hyperparameter combinations for all datasets, demonstrating that it is possible and beneficial to run evolutionary algorithms in an online way for the purpose of hyperparameter tuning.

Hyperon improved predictive performance across all datasets compared to Offline Grid Search and the Online Grid Search Upper Bound. Our results highlight the importance of changing the hyperparameter choices over time in data stream learning rather than using offline hyperparameter tuning to fix them at the beginning of the stream. They also suggest that Hyperon's ability to search for new hyperparameter configurations over time (rather than being restricted to a pre-defined grid of values) is helpful.

B. *RQ2: Hyperon's Computational Cost*

Hyperon is designed as an online hyperparameter tuning approach, where only a single iteration of the evolutionary algorithm is executed at every $eaInterval$ time steps. Hence, its computational cost is expected to be much smaller than if a full evolutionary algorithm had been run at regular intervals. However, Hyperon still needs to train and track the performance of multiple models simultaneously to enable continuous learning and hyperparameter adaptation, including switching between hyperparameter combinations that are within the population between EA intervals. Due to this, its computational cost is naturally expected to be higher than that of a static Offline Grid Search. It is thus important to check how high this computational cost is, to understand whether Hyperon would be feasible enough to adopt in practice. Therefore, this section compares the computational cost of Hyperon with that of the Offline Grid Search run on the initial portion of the stream.

Table III presents the total average runtime of Grid Search and Hyperon in Columns 2 and 3, respectively, computed across 30 runs for each of the 10 data streams on an Intel(R) Xeon(R) CPU E5-2690 v3 at 2.60GHz and 16Gb of RAM. Columns 4–5 list the average runtime per day (total runtime divided by project duration in days). Column 6 shows Hyperon's

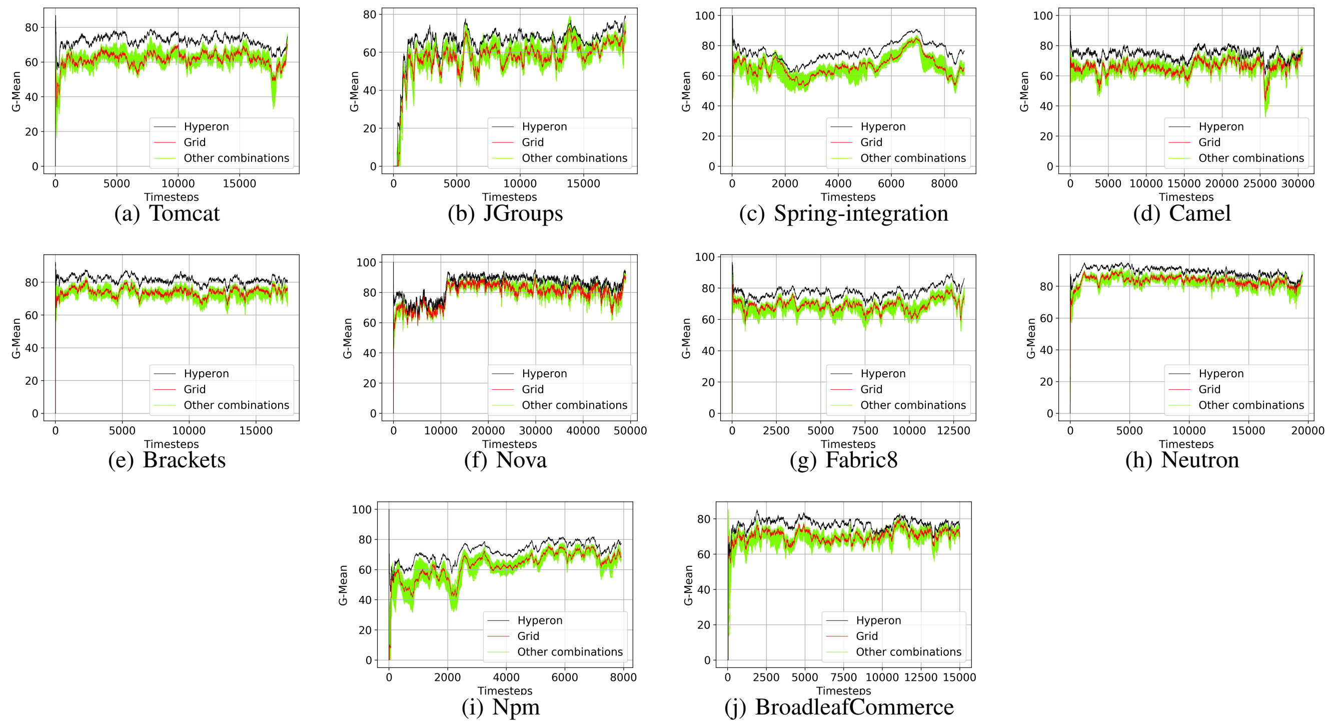

Fig. 1: G-Mean for all datasets through time using Hyperon (black), Offline Grid Search (red) and other 242 combinations (green).

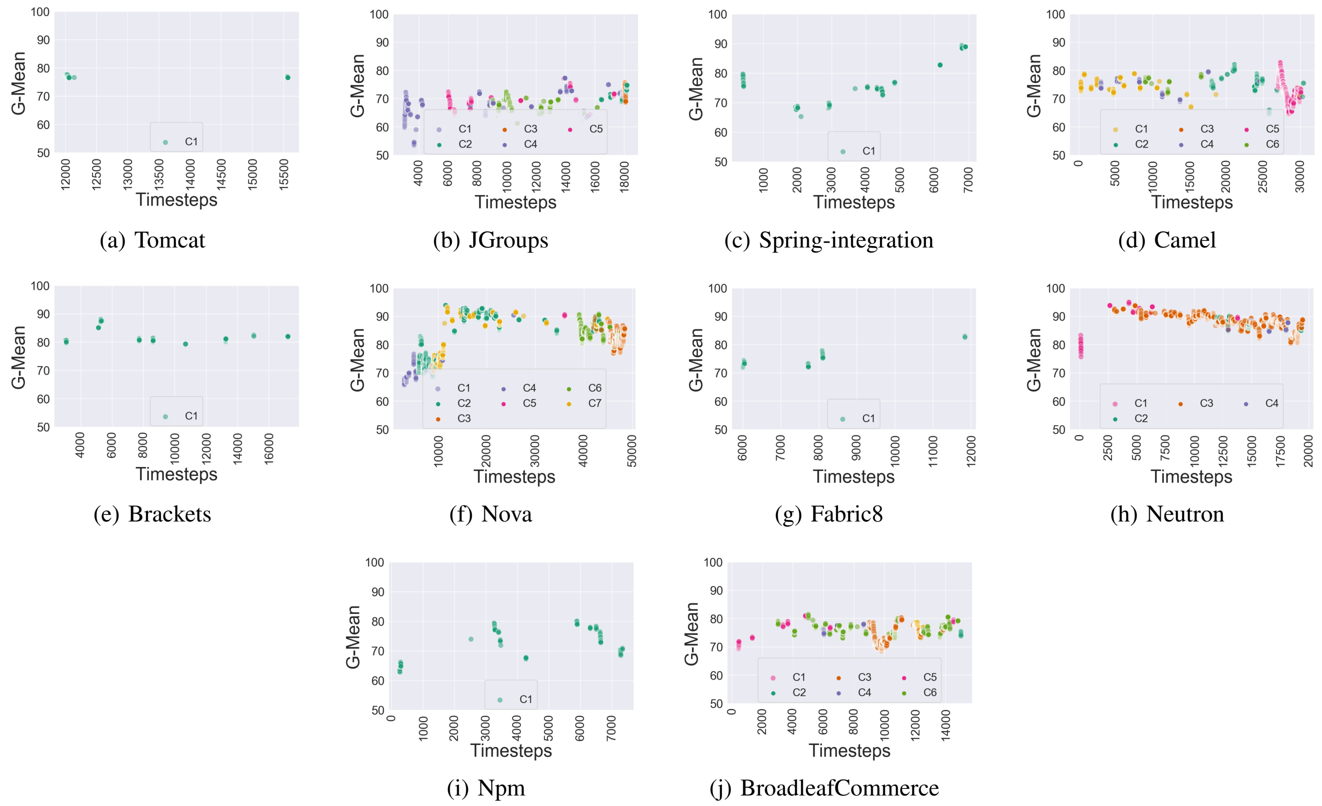

Fig. 2: New hyperparameter combinations discovered by Hyperon over time that obtained best g-mean. The point corresponding to a given hyperparameter combination is plotted when this combination is selected to be used for predictions and is omitted in the consecutive time steps where it is used, to improve readability of the plots.

TABLE II: Average G-Means of Hyperon, Grid search and Dummy classifier, A12 effect sizes, Scott-Knott.BA12 to compare Hyperon, Offline Grid Search and Dummy classifier.

Dataset	Approach	G-Mean	R0	R1
Tomcat	Hyperon	72.24 (0.25)	81.26 (0.63)	64.81 (0.7)
	Offline Grid	61.28 (0.57) [-b]	70.23 (0.87) [-b]	55.25 (0.6) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Jgroups	Hyperon	65.43 (1.07)	82.4 (1.15)	54.56 (2.14)
	Offline Grid	57.17 (0.8) [-b]	69.09 (1.46) [-b]	51.17 (0.99) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Spring-Integration	Hyperon	75.73 (0.24)	81.08 (0.38)	71.82 (0.56)
	Offline Grid	66.6 (0.88) [-b]	71.91 (0.81) [-b]	63.51 (1.51) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Camel	Hyperon	74.36 (0.38)	78.96 (0.61)	70.55 (1.09)
	Offline Grid	65.78 (0.57) [-b]	67.56 (0.81) [-b]	65.35 (0.6) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Brackets	Hyperon	81.51 (0.17)	81.02 (0.37)	82.31 (0.47)
	Offline Grid	73.77 (0.39) [-b]	72.5 (0.87) [-b]	75.6 (0.88) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Nova	Hyperon	85 (0.15)	80.44 (0.28)	90.34 (0.45)
	Offline Grid	78.97 (0.15) [-b]	74.56 (0.37) [-b]	84.53 (0.5) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Fabric8	Hyperon	77.37 (0.24)	79.8 (0.33)	75.26 (0.61)
	Offline Grid	69.01 (0.48) [-b]	71.09 (0.62) [-b]	67.61 (1.0) [-b]
	Dummy	71.59 [-b]	76.4 [-b]	81.64 [-b]
Neutron	Hyperon	89.09 (0.17)	86.55 (0.37)	91.83 (0.54)
	Offline Grid	83.36 (0.24) [-b]	80.76 (0.56) [-b]	86.3 (0.53) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Npm	Hyperon	71.96 (0.39)	76.36 (0.46)	68.97 (0.98)
	Offline Grid	62.64 (0.91) [-b]	62.42 (1.65) [-b]	66.51 (1.71) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Broadleafcommerce	Hyperon	76.76 (0.64)	81.55 (0.63)	72.84 (1.52)
	Offline Grid	70.32 (0.58) [-b]	72.33 (0.51) [-b]	69.05 (0.9) [-b]
	Dummy	50.00 [-b]	50.00 [-b]	50.00 [-b]
Ranking	Hyperon	1	1	1
	Offline Grid	2	2	2
	Dummy	3	3	3

Standard deviations are shown in brackets. Symbols [*], [s], [m] and [b] represent insignificant, small, medium and large A12 effect size against the Hyperon approach. Presence/absence of the sign "-" in the effect size means that the corresponding approach was worse/better than the Hyperon approach. Scott-Knott.BA12 was run for all approaches together. The groups' rankings retrieved by Scott-Knott.BA12 are shown in the ranking rows, with smaller numbers indicating better rankings.

TABLE III: Average Computational Cost for Hyperon and Offline Grid search

Dataset	Offline Grid Runtime (s)	Hyperon Runtime (s)	Offline Grid Runtime Per Day (s)	Hyperon Runtime Per Day (s)	Hyperon Runtime Per Training Example (ms)	# Training Examples After Target Project Starts	Data Stream Duration (years)
Tomcat	1345.37	74077.76	0.32	17.35	6.067	221752	11.7
JGroups	1138.37	85753.47	0.22	16.49	5.0361	226042	14.25
Spring-integration	1152.73	70823.3	0.31	19.06	5.37707	214379	10.18
Camel	1195.2	96433.35	0.31	24.62	5.47822	218173	10.73
Brackets	1341.43	66864.03	0.61	30.53	8.33808	160880	6
Nova	1451.5	93467.23	0.52	33.34	7.55521	192119	7.68
Fabric8	1252.27	76679.72	0.52	31.59	7.14721	175211	6.65
Neutron	1274.23	81804.29	0.5	32.06	7.01441	181659	6.99
Npm	1085.07	65264.52	0.36	21.91	5.45634	198864	8.16
BroadleafCommerce	1346.53	79799.2	0.41	24.27	6.5336	206093	9.01

runtime per training example. Column 7 shows the number of training examples in the data stream after the target project starts, whereas the last column shows the duration of each project in years. Note that the number of training examples in Column 7 is larger than the number of examples from each individual stream (Table I), as AIO-ORB is based on online transfer learning from the different datasets [37]. The maximum runtime of Offline Grid Search and Hyperon were up to 0.61 and 33.34 seconds per day across the duration of the data streams. Though Hyperon's runtime was higher than that of Offline Grid Search, it was still very small. Hence, it is likely feasible enough for a variety of real-world applications. Hyperon's processing time per training example was at most 8.33808ms (Column 6). So, for applications with similar input feature sizes as the ones in this study, Hyperon would be applicable so long as the incoming rate of examples is not larger than 1 per 8.3308ms.

Hyperon had higher computational cost compared to Offline Grid Search. However, the computational cost was still very small (less than 9ms per training example), evidencing feasibility for practical deployment.

VIII. Conclusion

This study proposed a novel online hyperparameter tuning for data stream learning called Hyperon. Hyperon was able to dynamically discover and adopt new hyperparameter configurations throughout the data stream. It achieved up to 10.96% and 6.9% absolute improvements in G-Mean when compared with Offline Grid Search run on an initial portion of the data stream, and the maximum possible G-Mean obtained by Online Grid Search (Upper Bound). Although higher than that of Offline Grid search, Hyperon's computational cost was very small (less than 9ms per training example).

Our results are based on ten real world datasets. As with any machine learning study, results may not generalise to other datasets dissimilar to those used in the study. Future work may incorporate additional datasets, online learning models, baselines, and an ablation study to investigate the influence of each of Hyperon's components on its performance.

References

[1] S. Tabassum, "Online cross-project prediction of defect-inducing software changes," Ph.D. dissertation, University of Birmingham, 2022.

[2] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE TSE*, vol. 45, no. 7, pp. 683–711, 2018.

[3] H. M. Gomes, J. Read, A. Bifet, J. P. Barddal, and J. Gama, "Machine learning for streaming data: state of the art, challenges, and opportunities," *ACM SIGKDD Explorations Newsletter*, vol. 21, no. 2, pp. 6–22, 2019.

[4] S. Tabassum, L. L. Minku, D. Feng, G. G. Cabral, and L. Song, "An investigation of cross-project learning in online just-in-time software defect prediction," in *2020 IEEE/ACM ICSE*, 2020, pp. 554–565.

[5] T. Paladini, M. Bernasconi de Luca, M. Carminati, M. Polino, F. Trovò, and S. Zanero, "Advancing fraud detection systems through online learning," in *ECML PKDD*, 2023, pp. 275–292.

[6] S. Hirsch, "Online multivariate regularized distributional regression for high-dimensional probabilistic electricity price forecasting," *arXiv preprint arXiv:2504.02518*, 2025.

[7] H. Ren, D. Anicic, X. Li, and T. Runkler, "On-device online learning and semantic management of TinyML systems," *ACM TECS*, vol. 23, no. 4, pp. 1–32, 2024.

[8] L. Song, L. L. Minku, and X. Yao, "The impact of parameter tuning on software effort estimation using learning machines," in *PROMISE*, 2013, pp. 1–10.

[9] B. Veloso, J. Gama, B. Malheiro, and J. Vinagre, "Hyperparameter self-tuning for data streams," *Information Fusion*, vol. 76, pp. 75–86, 2021.

[10] L. Kidane, P. Townend, T. Metsch, and E. Elmroth, "Automated hyperparameter tuning for adaptive cloud workload prediction," in *IEEE/ACM UCC*, 2023, pp. 1–8.

[11] I. A. Lawal and S. A. Abdulkarim, "Adaptive SVM for data stream classification," *SACJ*, vol. 29, no. 1, pp. 27–42, 2017.

[12] C. Lin, M. Guo, C. Li, X. Yuan, W. Wu, J. Yan, D. Lin, and W. Ouyang, "Online hyper-parameter learning for auto-augmentation strategy," in *IEEE/CVF ICCV*, 2019, pp. 6579–6588.

[13] L. L. Minku, "A novel online supervised hyperparameter tuning procedure applied to cross-company software effort estimation," *EMSE*, vol. 24, no. 5, pp. 3153–3204, 2019.

[14] R. Jie, J. Gao, A. Vasnev, and M.-N. Tran, "Hypertube: A framework for population-based online hyperparameter optimization with resource constraints," *IEEE Access*, vol. 8, pp. 69 038–69 057, 2020.

[15] T. T. Joy, S. Rana, S. Gupta, and S. Venkatesh, "Hyperparameter tuning for big data using bayesian optimisation," in *ICPR*, 2016, pp. 2574–2579.

[16] G. G. Cabral, L. L. Minku, E. Shihab, and S. Mujahid, "Class imbalance evolution and verification latency in just-in-time software defect prediction," in *IEEE/ACM ICSE*, 2019, pp. 666–676.

[17] S. Wang, L. L. Minku, and X. Yao, "A systematic study of online class imbalance learning with concept drift," *IEEE TNNLS*, vol. 29, no. 10, pp. 4802–4821, 2018.

[18] P. Liashchynskyi and P. Liashchynskyi, "Grid search, random search, genetic algorithm: A big comparison for NAS," *arXiv preprint arXiv:1912.06059*, 2019.

[19] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *JMLR*, vol. 13, no. 2, 2012.

[20] D. Maclaurin, D. Duvenaud, and R. Adams, "Gradient-based hyperparameter optimization through reversible learning," in *ICML*, 2015, pp. 2113–2122.

[21] F. Pedregosa, "Hyperparameter optimization with approximate gradient," in *ICML*, 2016, pp. 737–746.

[22] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[23] J. Bergstra, B. Komer, C. Eliasmith, D. Yamins, and D. D. Cox, "Hyperopt: a python library for model selection and hyperparameter optimization," *IOPscience*, vol. 8, no. 1, p. 014008, 2015.

[24] M. Pelikan, D. E. Goldberg, E. Cantú-Paz *et al.*, "BOA: The bayesian optimization algorithm," in *GECCO*, 1999.

[25] J. Wu, X.-Y. Chen, H. Zhang, L.-D. Xiong, H. Lei, and S.-H. Deng, "Hyperparameter optimization for machine learning models based on bayesian optimization," *JEST*, vol. 17, no. 1, pp. 26–40, 2019.

[26] K. Jamieson and A. Talwalkar, "Non-stochastic best arm identification and hyperparameter optimization," in *AISTATS*, 2016, pp. 240–248.

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

[28] H. Zhan, G. Gomes, X. S. Li, K. Madduri, and K. Wu, "Efficient online hyperparameter optimization for kernel ridge regression with applications to traffic time series prediction," *arXiv:1811.00620*, 2018.

[29] S. Bera and I. Mukherjee, "Performance analysis of nelder-mead and a hybrid simulated annealing for multiple response quality characteristic optimization," in *IMECS*, 2010, pp. 1728–1732.

[30] K. I. McKinnon, "Convergence of the Nelder–Mead simplex method to a nonstationary point," *SIAM Journal on optimization*, vol. 9, no. 1, pp. 148–158, 1998.

[31] F. Gao and L. Han, "Implementing the nelder-mead simplex algorithm with adaptive parameters," *Computational Optimization and Applications*, vol. 51, no. 1, pp. 259–277, 2012.

[32] A. R. Moya, B. Veloso, J. Gama, and S. Ventura, "Improving hyperparameter self-tuning for data streams by adapting an evolutionary approach," *KDD*, vol. 38, no. 3, pp. 1289–1315, 2024.

[33] M. Zhabitsky and E. Zhabitskaya, "Asynchronous differential evolution with adaptive correlation matrix," in *GECCO*, 2013, pp. 455–462.

[34] F. Vavak and T. C. Fogarty, "Comparison of steady state and generational genetic algorithms for use in nonstationary environments," in *CEC*, 1996, pp. 192–195.

[35] A. Auger and N. Hansen, "A restart CMA evolution strategy with increasing population size," in *CEC*, 2005, pp. 1769–1776.

[36] J. Zhang and A. C. Sanderson, "Jade: adaptive differential evolution with optional external archive," *IEEE TEvC*, vol. 13, no. 5, pp. 945–958, 2009.

[37] S. Tabassum, L. L. Minku, and D. Feng, "Cross-project online just-in-time software defect prediction," *IEEE TSE*, vol. 49, no. 1, pp. 268–287, 2023.

[38] G. G. Cabral and L. L. Minku, "Towards reliable online just-in-time software defect prediction," *IEEE TSE*, vol. 49, no. 3, pp. 1342–1358, 2023.

[39] M. Shepperd and S. MacDonell, "Evaluating prediction systems in software project estimation," *IST*, vol. 54, no. 8, pp. 820–827, 2012.

[40] J. Gama, R. Sebastião, and P. Rodrigues, "On evaluating stream learning algorithms," *Machine Learning*, vol. 90, no. 3, pp. 317–346, 2013.

[41] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," *IEEE TSE*, vol. 45, no. 12, pp. 1253–1269, 2018.

[42] Q. Zhu, "On the performance of Matthews correlation coefficient (MCC) for imbalanced dataset," *Pattern Recognit. Lett.*, vol. 136, pp. 71–80, 2020.

[43] T. Menzies, Y. Yang, G. Mathew, B. Boehm, and J. Hihn, "Negative results for software effort estimation," *EMSE*, vol. 22, no. 5, pp. 2658–2683, 2017.

[44] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *JMLR*, vol. 7, pp. 1–30, 2006.