Efficient Online Automated Algorithm Selection in the Face of Data-Drift in Optimisation Problem Instances

Jeroen Rook*
Paderborn University
Paderborn, Germany
University of Twente
Enschede, The Netherlands
jeroen.rook@uni-paderborn.de

Quentin Renau*
Edinburgh Napier University
Edinburgh, Scotland, UK
q.renau@napier.ac.uk

Heike Trautmann
Paderborn University
Paderborn, Germany
University of Twente
Enschede, The Netherlands
heike.trautmann@uni-paderborn.de

Emma Hart
Edinburgh Napier University
Edinburgh, Scotland, UK
e.hart@napier.ac.uk

Abstract

In many real-world problems, instances arrive in a stream which is likely to experience drift in the instance space over time. If a classical algorithm selector is trained offline, i.e., on an initial part of the instance stream, downstream performance is often negatively impacted due to drift in the instance data. To overcome this limitation of classical algorithm selectors, we propose a novel online automated algorithm selection framework that first uses instance features to detect drift, and then periodically retrains a selector if drift occurs, ensuring continuity of performance in face of data-drift. To further improve both the effectiveness and efficiency of retraining, we also propose a process to continuously gather new training samples on the fly. Empirical comparison using a bin-packing scenario under three different drift scenarios shows that our framework is efficient in terms of the computational effort required to train a selector while maintaining good performance with respect to accuracy compared to several baselines.

Keywords

Online Automated Algorithm Selection, Drift detection, Parallel Algorithm Portfolios

ACM Reference Format:
Jeroen Rook, Quentin Renau, Heike Trautmann, and Emma Hart. 2025. Efficient Online Automated Algorithm Selection in the Face of Data-Drift in Optimisation Problem Instances. In *Foundations of Genetic Algorithms XVIII (FOGA '25), August 27–29, 2025, Leiden, Netherlands.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3729878.3746615

*Both authors contributed equally to this research.

FOGA '25, Leiden, Netherlands

ACM ISBN 979-8-4007-1859-5/2025/08
https://doi.org/10.1145/3729878.3746615

1 Introduction

A vast amount of literature from the machine-learning field documents evidence that streams of real-world data are subject to *data-drift*: a change in the statistical properties and characteristics of the input data over time. Comprehensive reviews of the state-of-the-art can be found in recent surveys such as [4, 8, 16]. A serious consequence of data-drift is that the performance of a trained machine-learning (ML) model will degrade over time if the input data encountered downstream deviates from the data the model was initially trained on. In most real-world scenarios, the direction of future drift is unknown at the time of creating the model and the only available training data comes from historical instances. Training on data that covers the entire space (even if it were available) is inefficient as most users are not interested in models that generalise to all possible data, but only the subset of the space relevant to them [5]. For these reasons, models cannot be guaranteed to generalise to future relevant scenarios. As a result, a great deal of effort within ML is directed towards drift-detection, i.e., establishing robust methods to recognise when data arriving in a stream is from another distribution than the one used to train.

Streaming data is also common in *optimisation* domains, where typically one needs to solve a stream of *instances* [5]. Like in ML, in most practical optimisation applications, the characteristics of instances in the stream can be expected to change over time: for example, the size or shape of items can change over time in packing problems due to product changes, while in routing, unforeseen events such as a pandemic can dramatically alter the characteristics of routing problems. Significant changes in the input data distribution may result in instances that lie in a different region of instance-space [32] to historically seen instances. In the simplest case when a single solver is being used to solve instances before the drift occurs, this could necessitate a change in solver to maintain performance. Alternatively, if a trained algorithm-selector is being used on a stream in which undergoes a significant shift in the input data distribution, then the performance of the selector is likely to degrade. However, algorithm-selection on streaming data has been highlighted as being an under-studied area of research

in optimisation [20]: specifically, we are unaware of any literature that studies data drift detection in streaming optimisation domains.

To address this, we propose a novel method to both detect and react to data-drift in a stream of instances. When drift is detected, the approach triggers a retraining of the selector to account for the new distribution of data in the stream. We use the online bin-packing domain as a proof-of-concept for our method as heuristics to solve this problem are deterministic and thus simplify the analysis. However, note that any problem domain with the following characteristics can be used with our method: (1) being able to compare the performance of algorithms across instances and (2) where there is a correlation between drift in instance features and algorithm performance.

The contributions are as follows:

- We demonstrate for the first time that we can reliably detect and react to drift in instance features in the bin packing domain that affect the choice of optimisation solver in three common data-drift scenarios;
- We propose a novel drift-guided algorithm selector which automatically updates the selection model when drift is detected: retraining the selector leverages data and information that has been continually acquired from the data stream and performance of the solvers, therefore does not impose any additional costs on the system in obtaining the information needed to retrain the solver.
- We test our methods against 7 baselines and further show that we obtain better performances than 5 of the baselines and similar performance to an algorithm selector that is continuously retrained as new instances arrive, regardless of whether drift is occurring (this selector represents the best practical selector for this problem but is unrealistic to use in practice), but using significantly less computational effort;
- We provide a dashboard enabling users to use our methods to other problem domains as well as explore variations on the scenarios in this paper. The code, data, additional plots, and the dashboard are available at [3].

In Section 2 we provide a high-level step by step description of our methods. Section 3 provides some background on online bin-packing, data-drift detection, and algorithm selection in streams of instances. Sections 4 and 5 presents the experiments conducted and their results while Section 6 presents the limitations of our method.

2 Motivation

As mentioned in Section 1, algorithm selector models in the literature are typically trained once using representative data and then deployed [30]: once in production, the model remains static, i.e, it is typically not updated after deployment. However, changes in the input data-distribution to the model can occur for multiple reasons arising from the external environment, which modifies the characteristics of the instances and therefore the choice of solver. In the best case, this can lead to selectors that deliver sub-optimal performance, while at worst, selectors can become completely unfit for purpose. To this end, we propose a novel method that is capable of both detecting changes in instance characteristics and automatically updating the selector model in response to new data, thereby creating robust algorithm selectors.

In this section, we describe our approach (Figure 1) step by step using a packing problem example. Consider a company that manufactures products from a catalogue and needs to pack them in a warehouse. Instances are represented by orders the company receives, with each order typically consisting of more than one product. Depending on the number and characteristics of items in each order, some packing heuristics may produce better results than others. Thus, an algorithm selector is trained on instances using features of the products of the current catalogue (e.g., size,shape, weights) and is then deployed.

Each day, for each new instance, features of the products needing to be packed are computed and fed into a drift detector. While instances contain items from the original catalogue (black instances in Figure 1), the drift detector does not detect drift and the pre-trained selector should suggest the best heuristic. However, at some later time, the company updates its catalogue with a new product, which has differences in its characteristics to existing products. New orders involving these new products lie in a different region of the instance space to previously encountered instances (blue instances in Figure 1). Features extracted from the new instances appear to be very different: drift is detected by the drift detector which triggers an update of the selector. As drift detection is a post-hoc analysis and requires more than one instance to be detected, there is a lag in the update of the selector. Drifting instances seen during this lag period are then used to retrain the selector. Once the selector is updated, it is re-deployed to select the optimal packing heuristic. In addition, since we now know the characteristics of instances involving the new product, the drift detector is also updated so that it will not trigger an update of the selector for future instances of the new product.

3 Background

3.1 Online Bin-Packing

An online bin-packing instance consists of a sequence of items which must be packed strictly in the order they arrive and no information about the sequence is known in advance of each item arriving. The goal is typically to minimise the number of bins. Simple packing heuristics that determine which bin each item should be placed in are surprisingly effective [15] such as:

- best-fit (BF) places each item into the feasible bin that minimises the residual space;
- first-fit (FF) places each item into the first feasible bin that will accommodate it;
- next-fit (NF) places the items into the current opened bin and if not possible, it closes the bin and opens a new bin;
- worst-fit (WF) places the item into that currently open bin into which it will fit with the most room left over.

The quality of the resulting packing O_{Fk} is defined by the commonly used Falkenauer metric [12] which returns a value between 0 and 1 where 1 is optimal and rewards packings that minimise empty space. This is calculated as the average, over all n bins, of the k^{th} power of 'bin efficiency', i.e. $\sum_{i=1}^{i=n} \frac{(fill_i/C)^k}{n}$, with C the capacity of the bins and $fill_i$ the sum of sizes of the items in bin i. Here we follow common practice and set $k = 2$.

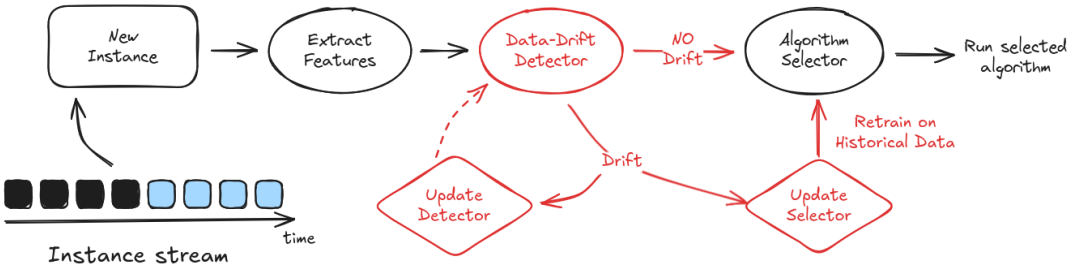


Figure 1: Automated Algorithm Selection pipeline using a drift detector. The drift detector is used to detect changes in instance features and triggers a retraining of the selector to account for these new unseen instances.

3.2 Data-Drift Detection

Data-drift is defined as a systematic shift in the underlying distribution of input features. It is also commonly referred to as *covariate* shift. It occurs when the characteristics of presenting data (e.g., samples in machine-learning, instances in optimisation) change over time — in most real-world applications, it is only a matter of time before it occurs. It differs from *concept-drift* where the relationship between the input and output may change over time, i.e., the same input features may lead to different labels over time. When using predictive models deployed in dynamic environments this is problematic as model performance can degrade due to the fact that the model was not trained/validated with data from the new distribution.

Data-drift detection methods fall into several classes [8, 23]: *statistical* (e.g., measuring change in the properties of a variable, e.g., its range); *distance-based*, measuring the distance between two probability distributions of data from different time-periods; *model-based* methods, which learn a model of 'normal' data and use this to detect anomalies. Methods can be further categorised as to whether they deal with univariate or multivariate data.

Univariate methods. These methods analyse drift in individual variables. Popular methods include a *statistical method* based on an incremental Kolmogorov-Smirnov test [10] or using a *distance-based metric* to measure differences between two distributions such as the Jensen-Shannon distance [22].

Multi-Variate Distance-based metrics. These methods aim to measure the distance between two probability distributions, the first representing data arriving in a *reference* period when no drift is assumed to occur, and the second in an *analysis* period in which drift may occur. A popular multivariate algorithm in this class is Maximum Mean Discrepancy (MMD) [29]. The algorithm calculates the MMD as a test statistic as the largest difference in expectations over functions in the unit ball of a reproducing kernel in a Hilbert space [17].

Data reconstruction methods. In this class of approaches, a data-compression model is trained on data obtained during a *reference* period, as described above. Any data-compression model of choice can be used, for example, PCA [25] or an AutoEncoder [35]. In the subsequent *analysis* period, new data is compressed using the previously learned model and then reconstructed. The magnitude of the reconstruction error between the original and reconstructed data indicates whether drift has likely occurred.

In this article, we select a data reconstruction method called NannyML [27] and use it off-the-shelf as a drift detector. Note that the goal of the paper is not to propose a novel method of drift detection but rather to study holistically how best to react to detected drift. Therefore any model that delivers reasonable detection performance could suffice. Initial empirical testing of NannyML showed promising accuracy on the bin packing datasets used as streams. Furthermore, the package has been used in a number of domains outside of optimisation to successfully monitor data-drift over time, e.g., in virus detection [13] and astronomical data [18].

The NannyML technique uses internally a PCA to learn the compressed model which makes the model cheap to execute. NannyML first applies PCA to data collected during the *reference period* to obtain a projection of this data to a 2d latent space. In subsequent chunks of data that arrive in the *analysis* period, each chunk is projected to the latent space using the learned PCA transformation, and then an inverse transformation is applied to recover the original data. The error between the original data points and those recovered from the inverse transformation is computed as the Euclidean distance between them. An *alert* indicating drift is generated if the data in a chunk in the analysis period has an absolute mean error that is more than three standard deviations from the mean error of the reference period. More information can be found in the NannyML documentation[1].

3.3 Algorithm Selection in Streams

Automated algorithm selection is usually formulated as a machine learning task where features describing the problem instance are used as input of a model while its output is used to select which algorithm from a portfolio should solve the instance [20]. Commonly, selectors are fitted on a representative instance set over which the features are computed, and all algorithms in the portfolio run on to get a target labelling on which algorithm performs best.

As mentioned in [20], many challenges arise for algorithm selection with streaming data: instances constantly arrive, their order cannot be modified and they have to be solved before carrying on with the stream. The underlying distribution of the instances may also change over time and may differ from the original distribution the selector has been trained on. Moreover, a challenge that arises

[1] https://nannyml.readthedocs.io/en/stable/

when deploying an algorithm selector in an online context is the lack of a ground-truth when only one algorithm is selected to run on an instance, especially on instances that are out of the initial distribution. Without running the other algorithms in the portfolio, the ground-truth – the best algorithm for the instance – cannot be determined. Consequently, maintaining a record where at any moment a new selector can be trained with becomes impractical.

Some studies on algorithm selection in the context of streaming data are available in the machine learning literature but typically focus on concept drift and/or selecting the classifier model which most improves accuracy of label detection over the stream. An approach called *MetaStream* is first proposed in [31] to select the best classifier for a given interval of observations, using an incrementally trained meta-classifier to select the best classifier in the case of *concept* drift. The approach is extended in [9] to actively select the best classifier for the current concept in a time-changing environment aided by an extended set of meta-features. A similar idea is proposed in [34], again to predict which classifier will perform best on a segment of a stream by also learning meta-features from the stream of data. Another approach in stream clustering, *confstream* [6], runs in parallel multiple clustering algorithms and multiple configurations to perform clustering on streaming data. In all these approaches, the motivation is to periodically replace the underlying classifier with a new model depending on the characteristics of the stream.

Algorithm selection on streaming data with a fixed type of classifier in which a trained classifier model outputs the most appropriate solver is an under-studied area of research in optimisation [20]. The problem differs from typical scenarios in ML which tend to focus on concept drift rather than data drift and on selecting the underlying classifier model itself. Within optimisation, a new benchmark for studying *concept* drift was recently proposed [36], enabling the generation of new streams of data which have different types of concept drift over time but does not consider *data* drift. The field of dynamic optimisation [2, 11] has also been well-studied but addresses changes in the *objective function* over time rather than changes in the data-distribution of instances and thus is different from the study we are conducting in this paper. We are unaware of any studies addressing detection and reaction to *data* drift in a stream of instances to update a model that selects the best solver by retraining on the fly with a subset of recent instances.

4 Experimental Setup

4.1 Online Bin-Packing Instances

Instances. We use a bin-packing dataset from Alissa *et. al.* [1]. The dataset is composed of 4,000 instances in which item sizes are generated from a uniform distribution in the range $(20, 100)$. Each instance is composed of 120 items and bins have a maximum capacity of 150. Instances have been evolved such that every heuristic presented in Section 3, BF, FF, NF, or WF is the winner of exactly 1,000 instances, according to the metric O_{Falk}. This can be observed in the 2d projection using UMAP [26] in Figure 2.

Features. As human-designed bin-packing features tend not to lead to high-performing algorithm selectors (see [1]) and online bin-packing has a temporal component, we treated each instance as a time-series and extracted time-series features.

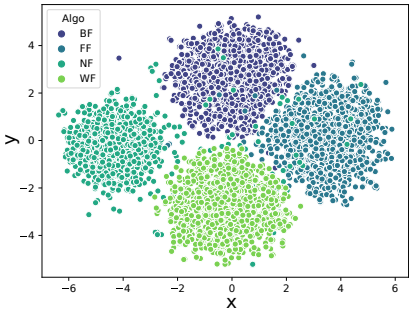


Figure 2: 2d projection of raw instance data, coloured by winning algorithm.

We used the tsfresh Python package [7][2] to extract features and performed two rounds of feature selection. This choice is motivated by the number of features extracted by tsfresh, around 400 statistics on the time-series characteristics. As feature selection can also be impacted by data-drift, we decide to perform it across all instances. Feature selection is performed on an algorithm selection task, i.e., given a feature vector on an instance, find which heuristic should be used to solve this instance. The first round of feature selection aims at reducing the number of features in order to perform a more fine-grained selection in the second round. The first round of feature selection is performed using the Boruta Python package [21] and results in 97 features. The second round of feature selection is performed using recursive feature elimination using the scikit-learn [28] Python package until 10 features are left (this number was empirically found and further tuning can be applied).

4.2 Drifting Scenarios

We order the instances described in Section 4.1 to simulate three common drift scenarios [23], illustrated schematically in Figure 3 and described below. We omit the blip drift scenario from [23] as its definition corresponds to a very short, unpredicted change in instance data and then the stream returns to instances with similar properties to the initial stream. As the drift period is very short, detecting and reacting to it may not be useful.

- **Incremental:** The characteristics of the instances in the stream incrementally change over time. We generate this stream based on the performance of heuristics contained in the reference period, i.e., we sort instances based on their performance O_{Fk}, from best to worst;
- **Recurrent:** periodically, a large drift occurs which is maintained for some period of time, then the stream returns to instances with similar properties to the initial stream;
- **Sudden:** at time t, the stream changes so that *all* instances that come after t are very different than those before t.

4.3 Drift Generation and Monitoring

In this paper, we generated three streams covering the three presented scenarios: an incremental, a sudden and a recurrent drift.

[2] https://tsfresh.readthedocs.io/en/latest/index.html

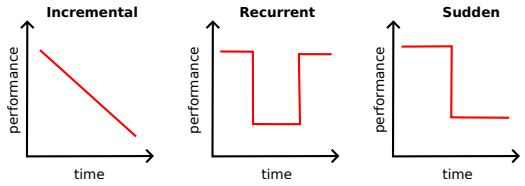


Figure 3: Diagram representing the three drifting scenarios: incremental, recurrent, and sudden.

The number of instances used in the reference period is always 500 while the number of instances in the analysis period may vary depending on the scenario. The chunk size is set to $c = 50$, i.e., the reconstruction error is computed every 50 instances. The chunk size value has great impacts on the detection and reaction to drift. Bigger values are more computation heavy, will prevent quick reactions to drift but may be more accurate in estimating the underlying distribution of instances as more instances will be used in the analysis. Smaller values enables quick reactions but may be prone to false positives or negatives as few instances are used during the analysis. In our experiments, we noticed that very small chunk size $c = 5$ produced false positives and false negative detections for every drift scenario while all values $c \geq 10$ produced similar drift detection results. We deliberately choose a chunk size $c = 50$ as it provides a good compromise with a sufficient number of instances and fairly fast possibility of reaction. Nevertheless, an in-depth study on the impact of this parameter on the selection results could be beneficial.

The incremental scenario uses WF and FF in the reference period while the sudden scenario uses BF, FF, and WF in the reference period. Finally, the recurrent scenario contains instances won by BF and FF in the reference period. The heuristic composing the reference period for all scenarios have to be chosen randomly. The recurrent scenario uses three periods composed of 200 instances during the stream where the instances are won by different heuristics. Between two drifting periods, we set 300 instances to be from the same distribution as the reference period. The number of instances in these periods have been chosen to be bigger than the chunk size to allow the possibility of detecting and reacting to drift. Any other number of instances bigger than the chunk size provides similar results. We repeat the three scenarios 50 times varying the order in which the instances are presented in the streams.

As mentioned in Section 3, we use NannyML to detect drift. NannyML can be used as a univariate or multivariate method. We choose the latter given the number of features used. A univariate drift detection would require an extra processing step to merge the drift detection for each feature.

4.4 Algorithm Selection Model

As mentioned in Section 3.3, algorithm selection in streams poses new challenges compared to static data. The main challenge is the lack of ground-truth for out-of-initial-distribution instances as no performance information on any algorithm from the portfolio is available. To address this, we integrate a continuous ground-truth collection procedure using parallel algorithm runs into the per-instance algorithm selection pipeline, i.e., alongside the selected

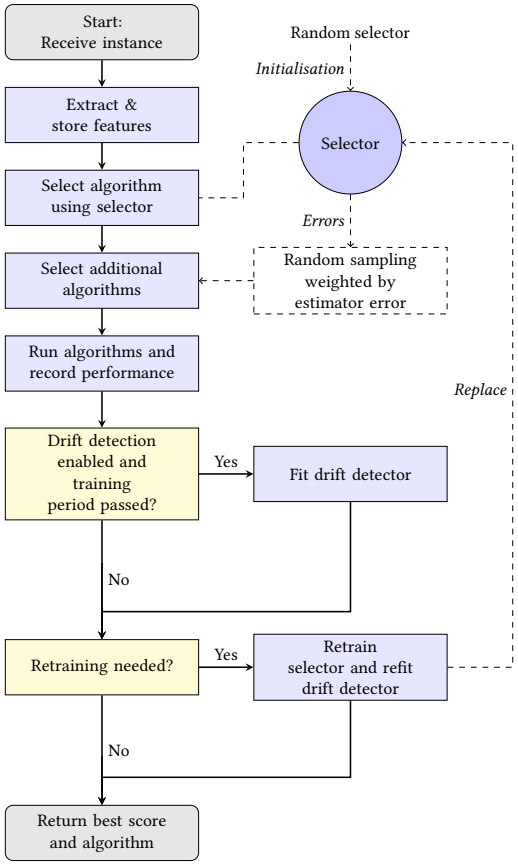


Figure 4: Flowchart describing the processes of ASDrift when a new instance is presented.

algorithm, we run another algorithm whose purpose is to help collect data. We furthermore present a special classifier fitting protocol that is compatible with this data collection procedure. In Figure 4, we illustrate the pipeline that is executed when new instances are presented to the selector. This pipeline starts with the traditional algorithm selector steps: compute the instance features, query a selector with those features, and get an algorithm to run. The consecutive steps are added by us and handle the selection of additional algorithm to run and checks that handle the (re)training of drift detectors and selectors. We will now describe in more detail how parallel runs are used to create a ground-truth, how the classifier learns from this, and how the online selector is initialised.

Parallel runs. In offline algorithm selection, a ground-truth is normally constructed by running all algorithms on an instance set. This gives, for each instance, a label depicting the best performing algorithm. In an online setting, such a ground-truth is impractical to construct: all algorithms in the portfolio need to run on that instance to find which one is the best. Moreover, it also means that there is no need for a selector as all algorithms are run on an instance. However, by running a subset of algorithms from the algorithm portfolio (minimum 2 out of the 4 heuristics) in parallel, we can obtain pairwise comparisons between those algorithms

that can then be used to train a classification model; giving us a comparative ground-truth between algorithms. By varying the algorithm pair that run in parallel on each instance, we construct a diverse set of pairwise algorithm comparisons.

To obtain a subset of algorithms, we complement the selected algorithm with additional (in our study 1) algorithms from the portfolio. All selected and additionally chosen algorithms then run in parallel on the instance and their respective performances are stored in a run history. The best performance out of these runs is returned to the user.

To choose complementary algorithms, we devised two heuristics. The first, simple, heuristic samples an algorithm uniformly at random. The second heuristic uses a weighted random sampling where the weights are inversely proportional to the selectors' accuracy between the selected algorithm and prospective complementary algorithm. The motivation behind this error-guided heuristic is that the level of discrimination between algorithms varies per pair and that gathering more evidence between difficult to distinguish pairs improves the selector performance.

One-vs-One Classifier. When treating the selector as a multi-class classification task, with inputs $\mathcal{X}$ and with m algorithms as labels $\mathcal{Y} = \{1, \ldots, m\}$, the goal is to learn a function $f : \mathcal{X} \to \mathcal{Y}$, where, for any $x \in \mathcal{X}$ the function $f(x)$ maps to an element in $\mathcal{Y}$, i.e., $f(x) \in \mathcal{Y}$. A one-vs-one classifier consists of $\binom{m}{2}$ binary classifiers, one for each algorithm pair (i, j) with $1 \leq i < j \leq m$. During inference, the feature vector $x \in \mathcal{X}$ is passed to all binary classifiers:

$$F = \left(f_{i,j}(x) \mid 1 \leq i < m - 1 \wedge i < j \leq m\right),$$

after which majority voting is applied to get the overall prediction.

Normally, a binary classifier for algorithms i and j would be trained on the points in the training data $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^{n}$ where their labels occur: $\mathcal{D}_{i,j} = \{(x, y) \in \mathcal{D} \mid y = i \vee y = j\}$.However, in the absence of a ground-truth in the labels, we modify this to

$$\mathcal{D}_{i,j} = \{(x, \mathcal{R}(x, i) \geq \mathcal{R}(x, j)) \in \mathcal{S} \mid \mathcal{R}(x, i) \neq \varnothing \wedge \mathcal{R}(x, j) \neq \varnothing\},$$

where $\mathcal{S}$ holds the instances seen in the stream and $\mathcal{R} : \mathcal{S} \times \mathcal{Y} \to \mathbb{R} \cup \varnothing$ the observed performances of algorithm runs. In case an algorithm i did not run on an instance x, $\mathcal{R}(x, i) \mapsto \varnothing$. That way, each binary classifier is trained on instances on which both algorithms actually ran on and thus pairwise comparison between the two exist. Importantly, the classifier does not make use of any information from the drift detector or in which type of drift scenario it can expect: it operates solely on the observed algorithm performances.

Our one-vs-one multi-class classifier supports any type of binary classifier, and in principle can even differ between algorithm pairs. For sake of simplicity, we use a decision tree as binary classifier in this paper. With more sophisticated binary classifiers, the performance can arguably only improve.

It can occur that, when the partial one-vs-one classifier is trained, there are algorithm pairs for which there are no pairwise comparisons in the training data. When this happens, a Bernoulli random process with $p = 0.5$, i.e., a uniformly random predictor, is used as classifier.

Initialisation. We assume that the algorithm selector has no prior knowledge before deployment. Similar to the drift monitoring (Subsection 4.3), we make use of a reference period where algorithm pairs to run are uniformly at random sampled, without replacement, from the portfolio. At the end of the reference period, a classifier is fitted along with the drift detector. Later, when the drift monitor detects drift, the selector is retrained on all previously seen data. Each time the selector is retrained, the drift detector is also retrained on the full history, assuming the newly trained selector can now handle all seen new instance types.

4.5 Algorithm Selection Baselines

We compare our algorithm selection method that reacts to drift to a set of baselines. We use both classical algorithm selection and stream-specific baselines. We also compare our approach to a baseline in which a heuristic is randomly selected.

Virtual Best Solver (VBS) and Single Best Solver (SBS). As in classical algorithm selection [20], we compare our method to the virtual best solver, i.e., an oracle always selecting the best solver to solve the instance at-hand. In our streaming context, we define the single best solver as being the best solver on average during the reference period.

Double Best Solvers (DBS). Since we run two algorithms in parallel for online algorithm selection, we also include a static baseline including two algorithms. We coin Double Best Solvers (DBS) the pair of algorithms that gave the best mean performance during the reference period, i.e., for each instance, the performance of the DBS is the highest score from the algorithms in the pair.

Classical selector. We compare our method to a classical algorithm selection approach. We train a random forest classifier using the reference period as training instances. We chose the random forest since it – just like our one-vs-one classifier – also consists of decision trees. We consider that all data for instances in the reference period is known in order to train the selector. Even though this is not realistic in a streaming context, this approach shows how classical algorithm selection would perform in a streaming context where drift is not accounted for.

Fixed retrain selector. In a streaming context with drift, a fixed retrain selector is the best trained selector possible as it is constantly updated using all the historical data. Nevertheless, this selector is only a baseline and would be hard to use in practice as its retraining time would constantly increase due to the large amount of data seen in long streams. We train an algorithm selector specifically tailored to the streaming context where we train a one-vs-one classifier. The selector is retrained after every chunk, i.e., 50 in this paper, with all the previously obtained data.

Multi-Armed Bandit (MAB). We also use a contextual Multi-Armed Bandit [24] as baseline. This MAB uses the same time series features as the drift detector and is initialized after the reference period. During the reference period, solvers are chosen at random. We use the MABwiser [33] package with the recommended configuration of using the UCB1 learning policy and a radius neighbourhood policy of 5. For every instance, the two algorithms with the highest expected performance are selected to run. There are two variants that differ in the updating interval. One follows the fixed retrain interval of 50 and the other retrains only when there is drift detected.

Respectively, these variants are referred to as MAB-retrain50 and MAB-driftguided.

Random. This baseline selects 2 different solvers from the portfolio uniformly at random.

5 Results

In this section, we present results on three drift scenarios: incremental, sudden, and recurrent. For each scenario, 50 random orderings of instances have been performed for the sudden and recurrent scenarios. Only one ordering of the incremental scenario is available as instances are sorted with decreasing performances of the heuristics used in the reference period. Top and middle figures from Figure 5 show one of the 50 orderings: the instance composition of the stream (top) and the reconstruction error from the drift monitoring (middle). The bottom figure aggregates the algorithm selection results from all orderings and shows the rolling average (window size 100) of the selector accuracy in running the best algorithm. Our primary performance measure for the subsequent analysis is this selector accuracy during drift phases as it directly reflects the method's ability to perform when the stream is impacted by drift.

5.1 Incremental Drift

Instances in the incremental scenario are sorted in decreasing performance order of the two heuristics used in the reference period, i.e., FF and WF, such that

$$\max\left(O_{Fk}(FF_i), O_{Fk}(WF_i)\right) \geq \max\left(O_{Fk}(FF_{i+1}), O_{Fk}(WF_{i+1})\right),$$

with i an instance and $O_{Fk}(FF_i)$ the Falkenauer fitness for a packing done with FF (resp. for WF). Thus, for each instance in the stream, the maximum performance of the selected pair or heuristic decreases over time. The top figure of Figure 5a shows the distribution of instances by winning heuristic in the stream. The first 2,000 instances in the stream are won by either FF or WF while the remaining 2,000 instances are won by either BF or NF.

Drift detection. The middle figure of Figure 5a shows the reconstruction error computed with NannyML. The first 500 instances compose the reference period. The reconstruction error thresholds are derived from the mean reconstruction error obtained in the reference period, i.e., plus and minus three times the mean reconstruction error. Any chunk of instances with a reconstruction error within these thresholds is labelled non-drifting (blue lines in Figure 5a). Any chunk of instances with a reconstruction error above or below these thresholds is labelled as drifting and an alert is raised (red lines in Figure 5a).

A perfect drift detection for this scenario should raise no alerts on instances between 500 (instance 0 of the analysis period) and 2,000 as these instances are still won by heuristics composing the reference period. We observe that the drift detection in Figure 5a is 84.3% accurate, i.e., 59 of the 70 chunks are correctly identified as drifting or non-drifting. On the 11 mis-detected chunks of instances, 2 raise false positives, i.e., raised an alert when it should not, and 9 chunks show false negatives, i.e., no alert is raised when it should be. Interestingly, the 9 false negatives all happen between instance 2,000 and instance 2,500 of the stream, i.e., when the first instances won by other heuristics are in the stream. Even though the reference heuristics do not deliver the best performance on these instances, their performance may still be good given how the stream is constructed, i.e., instances arrive in decreasing order of performance of the heuristics in the reference period. Drift is consistently detected afterwards for all remaining chunks. We observe this delay in drift detection for all combinations of heuristics used in the reference period.

Algorithm selection. The rolling average of the selectors' accuracy in selecting the best solver at the bottom of Figure 5a reveals that before drift occurs, the MAB shows strong performance in selecting the best solver. Also the two drift variants and the fixed retrain selectors show that they are able to make correct predictions. This suggests that the continual acquisition of training data works well.

When the drift occurs all selectors are negatively impacted in their performance, especially between instance order 2,000 and 3,000 where BF occurs most frequently. Logically, the SBS and DBS drop to a score of 0 since none of the algorithms in the reference period occur any more. The fixed-retrain selector shows a gradual recovery of its performance. For the drift variants, this only occurs when the NF-favoured instances are more frequent. This is due to a drift monitoring alert, triggering a retraining of the selector.

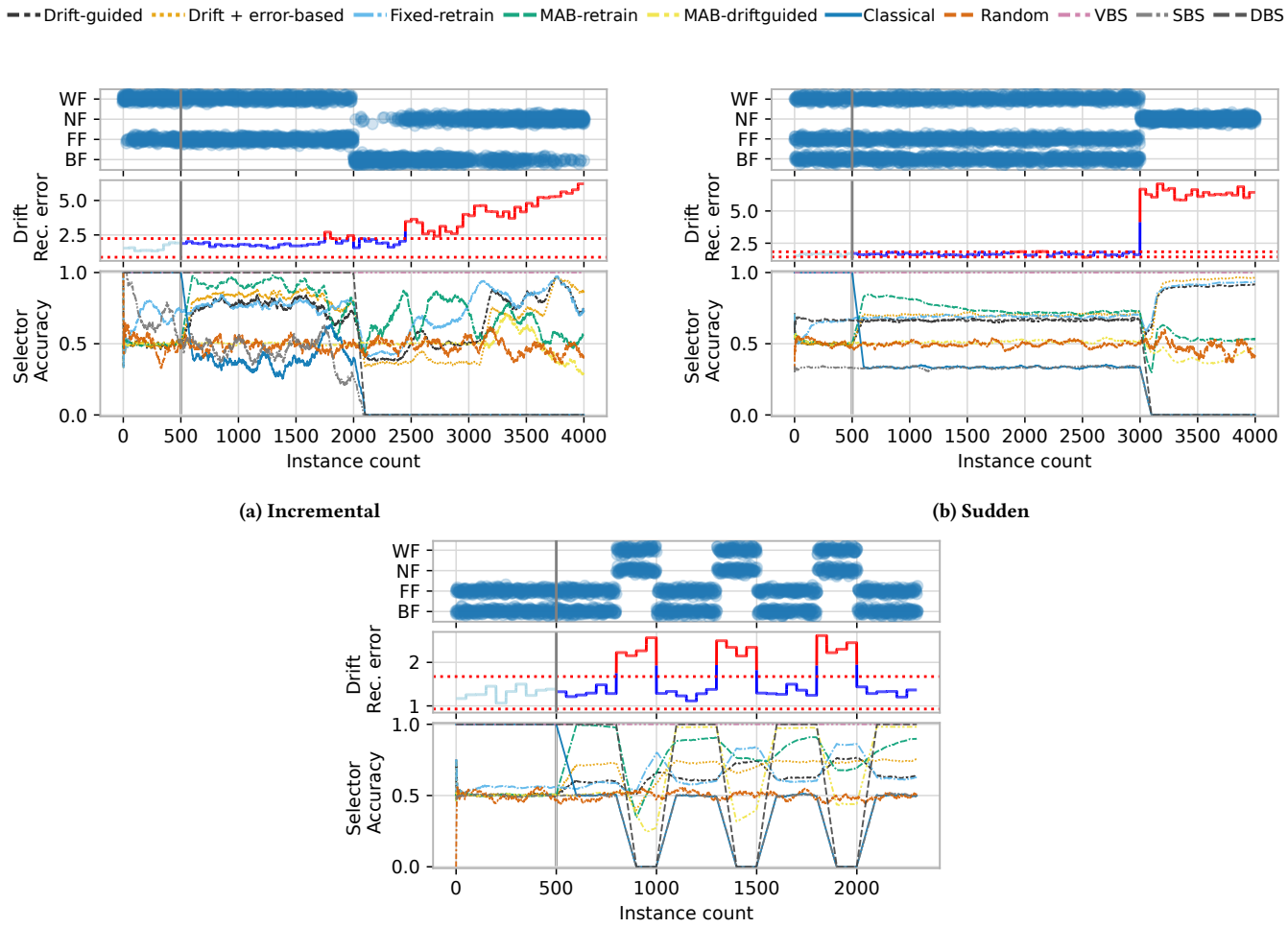
Finally, we observe that the error-based complementary run heuristic outperforms the selector with the (simple) random heuristic when there is no drift, however, with drift, it starts to reduce its performance. This indicates that there might be an overfitting effect on the classifier due to the classifier errors being static after training the classifier. In other words, the error-based heuristic does not account for the additional data that has been collected since the classifier was last trained. For this scenario, the drift-guided selector therefore is the recommended selector to use.

5.2 Sudden Drift

Instances in the sudden scenario are ordered such that all instances won by the heuristics in the reference period are first in the stream followed by all instances won by heuristics not present in the reference period. Figure 5b shows the distribution of instances by winning heuristic in the stream for a scenario with heuristics in the reference period being BF, FF, and WF. This ordering of instances is typical of the 50 we performed.

Drift detection. The middle figure of Figure 5b shows the reconstruction error computed with NannyML. The first 500 instances compose the reference period and are used to compute thresholds in the reconstruction error.

An ideal drift detection for this scenario should raise no alerts on instances between 500 (instance 0 of the analysis period) and 3,000 as these instances are still won by heuristics composing the reference period. An alert should consistently be raised afterwards as instances are won by another heuristic. We observe in Figure 5b that drift is detected across *all* chunks when the sudden change in instances appears. Out of the 50 runs of this scenario, we obtain no false positive alerts, i.e., the drift is always well detected. Nevertheless, as seen in the figure, there are 4 false positive alerts for this run. The average number of false positive chunks over the 50 runs is 1.06 with only 15 runs exhibiting false negatives and the average



(a) Incremental

(b) Sudden

(c) Recurrent

Figure 5: Instances composing the stream (top), drift detection (middle), and rolling algorithm selection accuracy (bottom) on the streams for the three scenarios.

drift detection accuracy is 98.5%, highlighting a good overall drift detection for this scenario.

Algorithm selection. The drift-guided and fixed-retrain selectors are able to deal well with the drift in this scenario. Their performances briefly drop just after the sudden drift occurred. They all share the same one-vs-one classifier, which again suggests that the sparse data collection works well. Retraining the selector only once shortly after the sudden drift does not negatively impact the performance compared to the fixed retraining baseline. The MAB variants on the contrary, are particularly poor in adjusting to the drift and their performances drop to the random baseline.

5.3 Recurrent Drift

Instances in the recurrent scenario are ordered such that after every 300 instances, 200 instances won by other heuristics are introduced in the stream. Figure 5c (top) shows the distribution of instances by winning heuristic in the stream for a scenario with heuristics in the reference period being BF and FF. This ordering of instances is typical of the 50 we performed.

Drift detection. The middle figure of Figure 5c shows the reconstruction error computed with NannyML. The first 500 instances compose the reference period and set the thresholds in the reconstruction error.

An accurate drift detector for this scenario should raise alerts on instances in the chunks where we introduced instances won by heuristics not present in the reference period and not raise an alert otherwise. We observe in Figure 5c that drift is detected in all chunks when the change in instances appears for each of the three drifting periods. Out of the 50 runs of this scenario, as in the sudden scenario, we obtain no false positive alerts, i.e., the

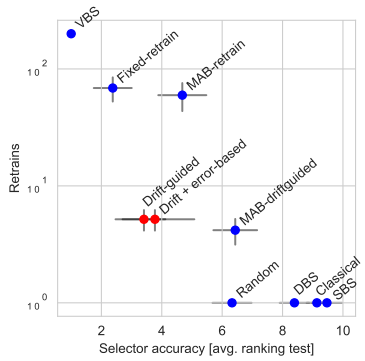
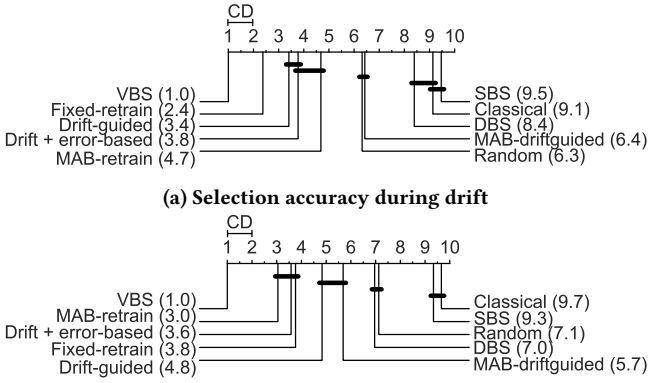


Figure 6: Trade-off between average number of retraining each selector and the ranking based on average selector accuracy when there is drift. Our proposed methods are depicted in red. The lines in each dimension represent the variance.



(a) Selection accuracy during drift

(b) Selection accuracy during complete test phase

Figure 7: Critical difference plots

drift is always well detected. Regarding false negative alerts, we observe in the figure one chunk where an alert is falsely raised. The average number of false positive chunks over the 50 runs is 0.3, resulting in a 99.2% accuracy in drift detection. Moreover, only 10 runs display a non-zero number of false negatives, highlighting a nearly perfect drift detection for this scenario. As the recurrent scenario is a succession of short sudden scenarios, this result could have been expected given results in Figure 5b.

Algorithm selection. The selector performances paint a similar picture with regard to their behaviour as with the other two scenarios. From the bottom part of Figure 5c, we see that the performance of the selectors drops directly after drift, but the drop in performance becomes less pronounced over time. The fixed-retrain selector actually stabilizes in performance towards the end, as can be seen in the bottom of Figure 5c. This suggests that retraining frequently helps for these drift types. Fortunately, due to the recurrent drift, the drift-guided and drift + error-based selectors also retrain regularly.

5.4 Overall performance

Retrain cost performance trade-off. One of the main motivations to do drift-guided selection is to prevent unnecessary retraining of an algorithm selector. The trade-off plot in Figure 6 shows that our drift-guided selectors give competitive performance, while the number of retrain events is a magnitude less than the selectors that are retrained in a fixed interval and they are positioned around the *knee* of the trade-off between retraining cost and performance.

Average rankings. To assess the overall effectiveness of the selectors, we compared the average rankings across all runs in the three scenarios ($3 \times 50 = 150$ runs). Figure 7 shows the critical difference plots on the selector accuracy for both the parts of the test set where there is drift (7a) and on the complete test set (7b). The critical difference (CD) is determined using the Nemenyi post-hoc test, with an alpha of 0.05 and is for both plots 0.98. When the difference between two average rankings is less than the CD, they are statistically tied. In Figure 7a, fixed-retrain shows, as expected, best performance when there is drift, followed by our two drift variants.

Table 1: Mean percentage accuracy, standard deviation and rank of methods when there is drift for each scenario.

	Incremental	Recurrent	Sudden
Drift-guided	69.5%±2.2% (3)	69.2%±10.0% (3)	86.1%±2.6% (4)
Drift + error-based	63.1%±4.3% (5)	63.9%±11.4% (5)	89.6%±2.6% (2)
Fixed-retrain	78.8%±1.4% (2)	78.2%±2.0% (2)	87.9%±2.4% (3)
MAB-retrain50	65.1%±1.1% (4)	65.7%±2.6% (4)	53.6%±2.3% (5)
MAB-driftguided	49.5%±2.5% (6)	39.1%±10.7% (7)	42.0%±18.1% (7)
Classical	4.5%±0.7% (9)	2.0%±4.1% (9)	2.0%±2.9% (10)
Random-2	47.8%±0.0% (7)	50.0%±1.6% (6)	47.2%±0.4% (6)
VBS	100%±0.0% (1)	100%±0.0% (1)	100%±0.0% (1)
SBS	2.3%±0.0% (10)	2.0%±3.9% (10)	2.1%±3.2% (9)
DBS	8.8%±0.0% (8)	3.9%±7.8% (8)	4.2%±6.3% (8)

The MAB variants are shown to be less able to adapt themselves when there is drift. In particular, the MAB with drift-guided retraining shows similar performance to the random selector. Possibly, the more exploitative behaviour of the MAB of selecting the solver with the highest expectancy is the cause of this. Selectors that only used the instances in the reference period have the lowest average rankings. In a broader perspective of the performance during the complete test phase (Figure 7b), the rankings slightly change. Surprisingly, the MAB with fixed retraining shows strong performance, but is statistically tied with the drift + cost based selector and the fixed retrain selector. The drift-guided selector shows better performances when drift occurs in comparison to the other drift variant suggests that being more explorative makes online selectors more suitable for drift.

Performance per scenario. Table 1 decomposes the selector accuracy during drift by the three considered scenarios. The relative performance differences for the selectors are similar across the scenarios. In particular, the recurrent and incremental scenarios yield similar performances in the accuracy of the selectors. The sudden scenario shows different behaviour among the selectors. The classifier-based selectors show particularly strong performance here, whereas the MAB variants have worse performance in comparison to the other two scenarios. We attribute this to the fact that the classifiers are seemingly better able to handle the NF instances in comparison to the MABs. The drift-guided selectors tend to have

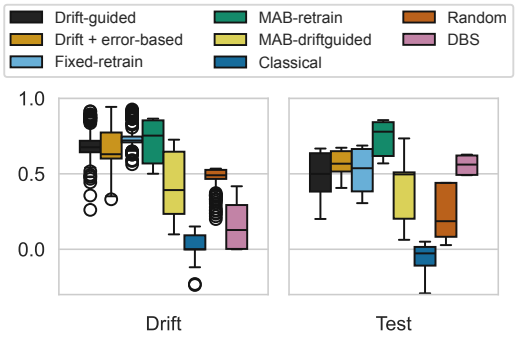


Figure 8: Boxplots of the percentage of the VBS-SBS gap that is closed by the selectors measured when drift occurs (left) and over the complete test set (right).

more variance in their performance, compared to selectors with a fixed training interval. We argue that the higher variance occurs because the drift-guided classifiers are trained only once after drift, whereas additional retraining improves prediction accuracy.

VBS-SBS gap. The VBS-SBS gap is the difference in the mean Falkenauer fitness between the SBS and VBS. Figure 8, provides an additional perspective on the actual performance of the selectors on the streams by showing how much of the SBS-VBS gap they close across all 150 runs. The boxplots show a similar trend compared to the findings when focussing on selector accuracy, where selectors that are only trained during the reference phase yield overall bad performance in the remainder of the instance stream, highlighting their inability to adapt to drift. While the MAB-retrain demonstrates relatively better performance on the complete test set, this difference becomes neglectable – 71% versus 69%, respectively – when only measuring the gap in phases with drift.

Discussion. In conclusion, the results demonstrate the effectiveness of drift-guided selectors in balancing retraining costs and performance. By significantly reducing the number of retraining events compared to selectors with fixed retraining, drift-guided selectors achieve comparative performance and position themselves near the optimal trade-off between cost and performance. While the fixed-retraining baseline shows strong performance, it is statistically tied with the drift selectors over the entire test phase. Interestingly, the MAB selectors, which tend to exploit the solvers with the highest expectancy, exhibit weaker performance during drift, suggesting that exploration plays an important role in the adaptability of selectors. These findings highlight the potential of drift-guided retraining with a sparse one-to-one classifier as an online algorithm selector in scenarios with data drift.

6 Limitations

In this work, we address the question of detecting and reacting to data-drift. However, our work has limitations and relies on some assumptions on the data. One of the limitations is the assumption that drift in instance features implies drift in algorithm performance, i.e., we assume that a change in instance features is correlated to a change in performance of a solver. Another possibility is that drift in instance features does not affect performance of algorithms — the same algorithm can perform well in different regions of the instance space. Thus, a broader pipeline detecting drift should be able to account for both cases. Another limitation lies in the ability to directly compare algorithm performances between instances. To ensure that there is a decay in performance when drift is detected in instance features, we need a performance measure that is on the same scale for each instance. For example, it is not possible to use the length of the tour for TSP as a performance metric as a big number for the tour length may be close to the optimum on one instance while a smaller number may be further away from the optimum on another instance. One can still use our method as it relies on instance data to detect drift in features but cannot check whether drift in performance is also present. Another possibility would be to integrate an indicator similar to fitness-distance correlation [19] that would permit to compare directly between instances.

7 Conclusion

This paper addresses the problem of data-drift detection and of dynamically updating an algorithm selector in a streaming optimisation context, a topic that is rarely considered in the optimisation field despite its prevalence in machine-learning [4, 16]. Specifically the goal is to detect changes in the input distribution of instance data that arrives over time in a stream where drift in the data might indicate the current selector should be retrained.

We generated three common drift scenarios in the online bin-packing domain. To address these drifting scenarios, we proposed novel methods that incorporate a drift detector monitoring the distribution of instance features and raising alerts when data drifted from historical data used to train an algorithm-selector. We use these alerts as a trigger to retrain our selector using new data collected along the stream by running a second solver along the selected one. Through these parallel solver runs, we devised a modified formulation for a one-vs-one multi-class classifier capable of performing classification in the absence of ground-truth labels. We show that our method is able to detect drift in the three scenarios encountered and that our drift-guided retraining, with significantly lower training cost, achieves similar performances with a selector that retrains continuously.

Online bin-packing is used as a proof-of-concept domain in this paper but our method is not limited to this. Other domains, both combinatorial or continuous, can be adapted to address the requirements for our method outlined in Section 6. For example, in the TSP domain, time could be used as an objective measure. With the goal of extending our results to more scenarios and more problem domains, we provide a dashboard [3] that enables any user to upload custom data from a domain, to create drifting scenarios and experiment with our method in streaming data scenarios.

Further studies on the impact of parameters involved in the drift detection are envisaged such as finding a trade-off for the chunk size between the quality of estimation of the underlying distribution of instance and the speed of reaction to drift detection.

An obvious direction for future work is to improve our selector with more complex models as well as larger algorithm portfolios, i.e., our one-vs-one sparse classifier currently uses a decision tree as a model and may not be suited to gather the ground-truth for large

sets of algorithms. Extending the approach to more complex models may lead to better and more robust results. Moreover, investigating the parameters of NannyML and other drift-detection methods is also essential to obtain more robust results as they shape both training phase and when methods are retrained. Another future work direction is online algorithm configuration [14] that handles drift and integrate those methods with our presented work.

In this paper, we address multiple drifting scenarios in the same way, i.e, by retraining our selector using previously seen data, disregarding the type of drift scenario. Future work on the detection of the specific type of drift scenario could lead to proactive actions. For example, in the detection of recurrent drift, one may be able to sample more instances like those seen in the drifting period to prepare the selector for future drifting periods. Finally, further works to extend our methods to other types of drift, such as drift in instance features that do not correlate with performance drift, are intended.

Acknowledgements

Quentin Renau and Emma Hart are supported by funding from EPSRC award number(s): EP/V026534/1.

This publication is based upon work from COST Action CA22137 "Randomised Optimisation Algorithms Research Network" (ROAR-NET), supported by COST (European Cooperation in Science and Technology).

References

[1] Mohamad Alissa, Kevin Sim, and Emma Hart. 2019. Algorithm selection using deep learning without feature extraction. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 198–206.

[2] Joan Alza, Mark Bartlett, Josu Ceberio, and John McCall. 2023. On the elusivity of dynamic optimisation problems. *Swarm and evolutionary computation* 78 (2023), 101289.

[3] Anonymous. 2025. Efficient Online Automated Algorithm Selection in the Face of Drifting Optimisation Problem Instances: Supplementary materials for reproducibility, additional figures and dashboard. doi:10.5281/zenodo.14761463

[4] J.P. Barddal, H.M. Gomes, F. Enembreck, and B. Pfahringer. 2017. A survey on feature drift adaptation: Definition, benchmark, challenges and future directions. *Journal of Systems and Software* 127 (2017), 278–294.

[5] Y. Bengio, A. Lodi, and A. Prouvost. 2021. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research* 290, 2 (2021), 405–421.

[6] M. Carnein, H. Trautmann, A. Bifet, and B. Pfahringer. 2020. confStream: Automated Algorithm Selection and Configuration of Stream Clustering Algorithms. In *Learning and Intelligent Optimization - 14th International Conference, LION 14, Athens, Greece, May 24-28, 2020, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 12096)*. Springer, 80–95. doi:10.1007/978-3-030-53552-0_10

[7] M. Christ, N. Braun, J. Neuffer, and A. W. Kempa-Liehr. 2018. Time Series FeatuRe Extraction on basis of Scalable Hypothesis tests (tsfresh - A Python package). *Neurocomputing* 307 (2018), 72–77. doi:10.1016/J.NEUCOM.2018.03.067

[8] L. Clever, J.S. Pohl, J. Bossek, P. Kerschke, and H. Trautmann. 2022. Process-Oriented Stream Classification Pipeline: A Literature Review. *Applied Sciences* 12, 18 (2022), 9094. doi:10.3390/app12189094

[9] Jáder MC de Sá, Andre LD Rossi, Gustavo EAPA Batista, and Luís PF Garcia. 2021. Algorithm recommendation for data streams. In *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE, 6073–6080.

[10] Denis Moreira dos Reis, Peter Flach, Stan Matwin, and Gustavo Batista. 2016. Fast Unsupervised Online Drift Detection Using Incremental Kolmogorov-Smirnov Test. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 1545–1554. doi:10.1145/2939672.2939836

[11] Johann Dréo and Patrick Siarry. 2006. An ant colony algorithm aimed at dynamic continuous optimization. *Appl. Math. Comput.* 181, 1 (2006), 457–467.

[12] Emanuel Falkenauer. 1996. A Hybrid Grouping Genetic Algorithm for Bin Packing. *Journal of Heuristics* 2, 1 (1996), 5–30. doi:10.1007/BF00226291

[13] Z. Farooq, H. Sjödin, J.C. Semenza, Y. Tozan, M.O. Sewe, J. Wallin, and J. Rocklöv. 2023. European projections of West Nile virus transmission under climate change scenarios. *One Health* 16 (2023), 100509.

[14] T. Fitzgerald, Y. Malitsky, B. O'Sullivan, and K. Tierney. 2014. ReACT: Real-Time Algorithm Configuration through Tournaments. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014*. AAAI Press, 62–70. doi:10.1609/SOCS.V5I1.18314

[15] Michael R Garey and David S Johnson. 1981. Approximation algorithms for bin packing problems: A survey. In *Analysis and design of algorithms in combinatorial optimization*. Springer, 147–172.

[16] R.N. Gemaque, A.F.J. Costa, R. Giusti, and E.M. Dos Santos. 2020. An overview of unsupervised drift detection methods. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 10, 6 (2020), e1381.

[17] A. Gretton, K.M. Borgwardt, M.J. Rasch, B. Schölkopf, and A. Smola. 2012. A Kernel Two-Sample Test. *Journal of Machine Learning Research* 13, 25 (mar 2012), 723–773.

[18] A. Humphrey, W. Kuberski, J. Bialek, N. Perrakis, W. Cools, N. Nuyttens, H. Elakhrass, and PAC. Cunha. 2022. Machine-learning classification of astronomical sources: estimating F1-score in the absence of ground truth. *Monthly Notices of the Royal Astronomical Society: Letters* 517, 1 (2022), L116–L120.

[19] T. Jones and S. Forrest. 1995. Fitness Distance Correlation as a Measure of Problem Difficulty for Genetic Algorithms. In *Proceedings of the 6th International Conference on Genetic Algorithms, Pittsburgh, PA, USA, July 15-19, 1995*. Morgan Kaufmann, 184–192.

[20] P. Kerschke, H.H. Hoos, F. Neumann, and H. Trautmann. 2019. Automated Algorithm Selection: Survey and Perspectives. *Evolutionary Computation* 27, 1 (March 2019), 3–45. doi:10.1162/evco_a_00242

[21] Miron B. Kursa and Witold R. Rudnicki. 2010. Feature Selection with the Boruta Package. *Journal of Statistical Software* 36, 11 (2010), 1–13. doi:10.18637/jss.v036.i11

[22] J. Lin. 1991. Divergence measures based on the Shannon entropy. *IEEE Transactions on Information Theory* 37, 1 (1991), 145–151. doi:10.1109/18.61115

[23] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang. 2019. Learning under Concept Drift: A Review. *IEEE Trans. Knowl. Data Eng.* 31, 12 (2019), 2346–2363. doi:10.1109/TKDE.2018.2876857

[24] Tyler Lu, David Pal, and Martin Pal. 2010. Contextual Multi-Armed Bandits. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 9)*, Yee Whye Teh and Mike Titterington (Eds.). PMLR, 485–492. https://proceedings.mlr.press/v9/lu10a.html

[25] A. Maćkiewicz and W. Ratajczak. 1993. Principal components analysis (PCA). *Computers & Geosciences* 19, 3 (1993), 303–342.

[26] L. McInnes and J. Healy. 2018. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *CoRR* abs/1802.03426 (2018). arXiv:1802.03426 http://arxiv.org/abs/1802.03426

[27] NannyML 2023. NannyML (version 0.8.3). https://github.com/NannyML/nannyml. NannyML, Belgium, OHL..

[28] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[29] S. Rabanser, S. Günnemann, and Z. Lipton. 2019. Failing loudly: An empirical study of methods for detecting dataset shift. *Advances in Neural Information Processing Systems* 32 (2019).

[30] Q. Renau, J. Dréo, A. Peres, Y. Semet, C. Doerr, and B. Doerr. 2022. Automated algorithm selection for radar network configuration. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '22*, Jonathan E. Fieldsend and Markus Wagner (Eds.). ACM, 1263–1271. doi:10.1145/3512290.3528825

[31] André Luis Debiaso Rossi, André CPLF Carvalho, and Carlos Soares. 2012. Meta-learning for periodic algorithm selection in time-changing data. In *2012 Brazilian Symposium on Neural Networks*. IEEE, 7–12.

[32] K. Smith-Miles and M.A. Muñoz. 2023. Instance Space Analysis for Algorithm Testing: Methodology and Software Tools. *ACM Comput. Surv.* 55, 12, Article 255 (mar 2023), 31 pages. doi:10.1145/3572895

[33] Emily Strong, Bernard Kleynhans, and Serdar Kadioglu. 2021. MABWiser: Parallelizable Contextual Multi-armed Bandits. *Int. J. Artif. Intell. Tools* 30, 4 (2021), 2150021:1–2150021:19. doi:10.1142/S0218213021500214

[34] J.N. van Rijn, G. Holmes, B. Pfahringer, and J. Vanschoren. 2014. Algorithm selection on data streams. In *Discovery Science: 17th International Conference, DS 2014, Bled, Slovenia, October 8-10, 2014. Proceedings 17*. Springer, 325–336.

[35] W. Wang, Y. Huang, Y. Wang, and L. Wang. 2014. Generalized autoencoder: A neural network framework for dimensionality reduction. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*. 490–497.

[36] Y. Zhong, X. Wang, Y. Sun, and Y-J. Gong. 2024. SDDObench: A Benchmark for Streaming Data-Driven Optimization with Concept Drift. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2024, Melbourne, VIC, Australia, July 14-18, 2024*. ACM. doi:10.1145/3638529.3654063